Python Basics and Data analysis handbook and cheetsheet

Part 1 -- Basics

This Notebook consist of all the example solutions of the book `No Starch Python` topics under this book are

- Variables and Datatypes
- Lists and For loops
- If conditinal statements
- Dictionaries
- While Loops and inputs
- Functions
- Classes
- Using external Files
- Exceptions

Chapter 2: Variables and Datatypes

Variables

Rules to Naming and Using Variables

When you're using variables in Python, you need to adhere to a few rules and guidelines. Breaking some of these rules will cause errors; other guide□lines just help you write code that's easier to read and understand. Be sure to keep the following variable rules in mind:

- Variable names can contain only letters, numbers, and underscores. They can start with a letter or an underscore, but not with a number. For instance, you can call a variable message_1 but not 1_message.
- Spaces are not allowed in variable names, but underscores can be used to separate words in variable names. For example,greeting_message works, but greeting message will cause errors.
- Avoid using Python keywords and function names as variable names; that is, do not use words that Python has reserved for a particular pro□grammatic purpose, such as the word print. (See "Python Keywords and Built-in Functions" on page 489.)
- Variable names should be short but descriptive. For example, name is better than n, student_name is better than s_n, and name_length is better than length_of_persons_name.
- Be careful when using the lowercase letter l and the uppercase letter O because they could be confused with the numbers 1 and 0.

In [3]:
```
str = "This is Recall of Python Basics"
print(str)
```

```
This is Recall of Python Basics
```

In [4]:
```
str1 = "This is Python Jupyter Notebook"
print(str1)
str1 = "This is using Vscode"
print(str1)
```

```
This is Python Jupyter Notebook
This is using Vscode
```

Strings Data Type

String is simply a series of characters. Anything inside quotes is con□sidered a string in Python, and you can use `single or double quotes` around your strings like this:

In [6]:
```
name = "Tony Stark"
print(name + " is the richest person in mcu")
print("{} is the richest person in MCU".format(name))
```

```
Tony Stark is the richest person in mcu
Tony Stark is the richest person in MCU
```

In [13]:
```
print("Title case: {}".format(name.title()))
print("Upper case: {}".format(name.upper()))
print("lower case: {}".format(name.lower()))
print("Capitalize case: {}".format(name.capitalize()))
```

```
Title case: Tony Stark
Upper case: TONY STARK
lower case: tony stark
Capitalize case: Tony stark
```

In [16]:
```
author = "Albert Einstein"
print('"A Person who never made a mistake never tried anything new" by {}'.format(au
```

```
"A Person who never made a mistake never tried anything new" by Albert Einstein
```

In [19]:
```
author = "albert einstein"
message = "A Person who never made a mistake never tried anything new"
print('"{a}" by {b}'.format(a= message, b=author.title()))
```

```
"A Person who never made a mistake never tried anything new" by Albert Einstein
```

In [24]:
```
person_name = "  Tony   "
print("Original string:\t'{}'\nLeft Strip:\t'{}'\nRight Strip:\t'{}'\nNormal Strip:\
```

```
Original string:        '  Tony   '
Left Strip:     'Tony   '
Right Strip:    '  Tony'
Normal Strip:   'Tony'
```

Numbers Data type

Among its basic types, Python provides integers (positive and negative whole numbers) and floating-point numbers (numbers with a fractional part, sometimes called “floats” for economy). Python also allows us to write integers using hexadecimal, octal, and binary literals; offers a

complex number type; and allows integers to have unlimited precision (they can grow to have as many digits as your memory space allows). Table 5-1 shows what Python's numeric types look like when written out in a program, as literals.

Basic numeric literals

- 1234, –24, 0, 99999999999999 ---------> Integers
- 1.23, 1., 3.14e-10, 4E210, 4.0e+210 ---------> Floating-point numbers
- 0177, 0x9ff, 0b101010 ---------> Octal, hex, and binary literals in 2.6 python
- 0o177, 0x9ff, 0b101010 ---------> Octal, hex, and binary literals in 3.0 python
- 3+4j* 3.0+4.0j, 3J ---------> Complex number literals

In [26]:
```
print("Addition: 5+3 = {}\n\nSubtraction: 10-2 = {}\n\nMultiplication: 4*2 = {}\n\nD
```

```
Addition: 5+3 = 8

Subtraction: 10-2 = 8

Multiplication: 4*2 = 8

Division: 16/2 = 8.0
```

In [27]:
```
fav_num = 18
print("My Favarite number is {}".format(fav_num))
```

```
My Favarite number is 18
```

Chapter 3

Lists

What Is a List?

A list is a collection of items in a particular order. You can make a list that includes the letters of the alphabet, the digits from 0–9, or the names of all the people in your family. You can put anything you want into a list, and 38 Chapter 3 the items in your list don't have to be related in any particular way. Because a list usually contains more than one element, it's a good idea to make the name of your list plural, such as letters, digits, or names. In Python, square brackets ([]) indicate a list, and individual elements in the list are separated by commas. Here's a simple example of a list that contains a few kinds of bicycles:

syntax: `name of list = ["values"]`

Accessing Elements in a List

Lists are ordered collections, so you can access any element in a list by telling Python the position, or index, of the item desired. To access an element in a list, write the name of the list followed by the index of the item enclosed in square brackets. `The indexing postion starts for '0'`

syntax: `list_name[index]`

In [29]:
```
names = ["iron man", "captain america", "hulk", "black widow", "black panther", "spi
for a in names:
```

```
    print(a.title())
```

```
Iron Man
Captain America
Hulk
Black Widow
Black Panther
Spider Man
```

In [31]:

```
for n in names:
    print("Hey, How are You {}".format(n.title()))
```

```
Hey, How are You Iron Man
Hey, How are You Captain America
Hey, How are You Hulk
Hey, How are You Black Widow
Hey, How are You Black Panther
Hey, How are You Spider Man
```

In [32]:

```
lt = ["honda", "bugatti", "audi", "mercedes", "bmw"]
print("I would like to own a {} Motorcycle".format(lt[0].title()))
print("I wanna by a {} Car".format(lt[1].title()))
print("{} is a car with comfort".format(lt[2].title()))
print("You don't even know a shake in {} car".format(lt[3].title()))
print("{} is a best Racing Car".format(lt[4].upper()))
```

```
I would like to own a Honda Motorcycle
I wanna by a Bugatti Car
Audi is a car with comfort
You don't even know a shake in Mercedes car
BMW is a best Racing Car
```

In [38]:

```
guests = ["iron man", "captain america", "hulk", "black widow", "black panther", "sp
for g in guests:
    print("Hello, {} today I have dinner at my home, can you come".format(g.capitali
```

```
Hello, Iron man today I have dinner at my home, can you come
Hello, Captain america today I have dinner at my home, can you come
Hello, Hulk today I have dinner at my home, can you come
Hello, Black widow today I have dinner at my home, can you come
Hello, Black panther today I have dinner at my home, can you come
Hello, Spider man today I have dinner at my home, can you come
```

In [39]:

```
guestnc = guests.pop(-2)
print("The guest not came to dinner is {}\n".format(guestnc.capitalize()))
guests.append("Doctor Strange")
print("New guest list:\n{}".format(guests))
for g in guests:
    print("Hello, {} today I have dinner at my home, can you come".format(g.capitali
```

```
The guest not came to dinner is Black panther

New guest list:
['iron man', 'captain america', 'hulk', 'black widow', 'spider man', 'Doctor Strang
e']
Hello, Iron man today I have dinner at my home, can you come
Hello, Captain america today I have dinner at my home, can you come
Hello, Hulk today I have dinner at my home, can you come
Hello, Black widow today I have dinner at my home, can you come
Hello, Spider man today I have dinner at my home, can you come
Hello, Doctor strange today I have dinner at my home, can you come
```

In [40]:

```
guests.insert(0, "vision")
guests.insert(4, "scarlet witch")
```

```
guests.append("war machine")
for g in guests:
    print("Hello, {} today I have dinner at my home, can you come".format(g.capitali
```

```
Hello, Vision today I have dinner at my home, can you come
Hello, Iron man today I have dinner at my home, can you come
Hello, Captain america today I have dinner at my home, can you come
Hello, Hulk today I have dinner at my home, can you come
Hello, Scarlet witch today I have dinner at my home, can you come
Hello, Black widow today I have dinner at my home, can you come
Hello, Spider man today I have dinner at my home, can you come
Hello, Doctor strange today I have dinner at my home, can you come
Hello, War machine today I have dinner at my home, can you come
```

In [41]:

```
print(r"Original List of guest:")
for g in guests:
    print(g.capitalize())
while len(guests) > 2:
    gnc = guests.pop()
    print("Sorry {} the big table is canceled".format(gnc.capitalize()))
```

```
Original List of guest:
Vision
Iron man
Captain america
Hulk
Scarlet witch
Black widow
Spider man
Doctor strange
War machine
Sorry War machine the big table is canceled
Sorry Doctor strange the big table is canceled
Sorry Spider man the big table is canceled
Sorry Black widow the big table is canceled
Sorry Scarlet witch the big table is canceled
Sorry Hulk the big table is canceled
Sorry Captain america the big table is canceled
```

In [42]:

```
for g in guests:
    print("Hello, {} you can come to dinner".format(g.capitalize()))
```

```
Hello, Vision you can come to dinner
Hello, Iron man you can come to dinner
```

In [43]:

```
i = 1
while i <= 2:
    del guests[0]
    i+=1
print(guests)
```

```
[]
```

In [77]:

```
loc = ["london", "russia", "germany", "mexico", "california", "arizona"]
locs= []
for p in loc:
    p = p.capitalize()
    locs.append(p)
print("Original list \n{}".format(locs))
print("sorted list: \n{}".format(sorted(locs)))
print("\n{}".format(locs))
print("Reverse sorted list: \n{}".format(sorted(locs, reverse= True)))
print("\n{}".format(locs))
locs.reverse()
```

```
print("\n{}".format(locs))
locs.sort()
print("\n{}".format(locs))
locs.sort(reverse = True)
print("\n{}".format(locs))
```

```
Original list
['London', 'Russia', 'Germany', 'Mexico', 'California', 'Arizona']
sorted list:
['Arizona', 'California', 'Germany', 'London', 'Mexico', 'Russia']

['London', 'Russia', 'Germany', 'Mexico', 'California', 'Arizona']
Reverse sorted list:
['Russia', 'Mexico', 'London', 'Germany', 'California', 'Arizona']

['London', 'Russia', 'Germany', 'Mexico', 'California', 'Arizona']

['Arizona', 'California', 'Mexico', 'Germany', 'Russia', 'London']

['Arizona', 'California', 'Germany', 'London', 'Mexico', 'Russia']

['Russia', 'Mexico', 'London', 'Germany', 'California', 'Arizona']
```

In [79]:
```
print(len(locs))
```

```
6
```

Chapter 4

Working though the list

In [1]:
```
pizzas = ["macroni", "salami", "cheese burst"]
for p in pizzas:
    print(p.capitalize())
```

```
Macroni
Salami
Cheese burst
```

In [3]:
```
for p in pizzas:
    print("I like Dominos {} pizza".format(p.capitalize()))
print("I like {} pizza very much than other".format(pizzas[0].capitalize()))
```

```
I like Dominos Macroni pizza
I like Dominos Salami pizza
I like Dominos Cheese burst pizza
I like Macroni pizza very much than other
```

In [4]:
```
anim = ["dog", "cat", "fish"]
for a in anim:
    print(a.capitalize())
```

```
Dog
Cat
Fish
```

In [6]:
```
for a in anim:
    print("I like to have a pet like {}".format(a.capitalize()))
```

```
print("\n I love {} as a pet".format(anim[0].capitalize()))
```

```
I like to have a pet like Dog
I like to have a pet like Cat
I like to have a pet like Fish

 I love Dog as a pet
```

In [7]:
```
for n in range(21):
    print(n)
```

```
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
```

In [1]:
```
num = list(range(1,501))
print("{}.....{}".format(num[:6],num[-6:]))
```

```
[1, 2, 3, 4, 5, 6].....[495, 496, 497, 498, 499, 500]
```

In [89]:
```
max(num)
```

Out[89]: 500

In [90]:
```
min(num)
```

Out[90]: 1

In [91]:
```
sum(num)
```

Out[91]: 125250

In [15]:
```
odd = list(range(1,21,2))
for o in odd:
    print(o)
```

```
1
3
5
7
9
```

```
11
13
15
17
19
```

In [16]:

```
mul3 = list(range(3,31,3))
for m in mul3:
    print(m)
```

```
3
6
9
12
15
18
21
24
27
30
```

In [17]:

```
cubes = [c**3 for c in range(1,11)]
for cb in cubes:
    print(cb)
```

```
1
8
27
64
125
216
343
512
729
1000
```

In [18]:

```
print(cubes)
```

```
[1, 8, 27, 64, 125, 216, 343, 512, 729, 1000]
```

In [22]:

```
mylst = ["Vision", "Iron man", "Captain america", "Hulk", "Scarlet witch", "Black wi
print("The first three items in the list are :\n{}".format(mylst[:3]))
print("\n")
print("The Items in the middle of the list are: \n{}".format(mylst[3:6]))
print("\n")
print("The Items in the last of the list are: \n{}".format(mylst[6:]))
```

```
The first three items in the list are :
['Vision', 'Iron man', 'Captain america']


The Items in the middle of the list are:
['Hulk', 'Scarlet witch', 'Black widow']


The Items in the last of the list are:
['Spider man', 'Doctor strange', 'War Machine']
```

In [37]:

```
pizzas = ["macroni", "salami", "cheese burst"]
friends_pizzas = pizzas[:]
pizzas.append("burger pizza")
friends_pizzas.append("chiken")
print("My pizzas:")
for p in pizzas:
```

```
    print(p.capitalize())
print("\nMy friends pizzas:")
for fp in friends_pizzas:
    print(fp.capitalize())
```

```
My pizzas:
Macroni
Salami
Cheese burst
Burger pizza

My friends pizzas:
Macroni
Salami
Cheese burst
Chiken
```

Tuples

Lists work well for storing sets of items that can change throughout the life of a program. The ability to modify lists is particularly important when you're working with a list of users on a website or a list of characters in a game. However, sometimes you'll want to create a list of items that cannot change. Tuples allow you to do just that. Python refers to values that cannot change as immutable, and an immutable list is called a tuple.

Defining a Tuple

A tuple looks just like a list except you use `parentheses instead of square brackets`. Once you define a tuple, you can access individual elements by using each item's index, just as you would for a list.

In [40]:
```
menu = ("idly", "dosa", "pongal")
for m in menu:
    print(m.capitalize())
```

```
Idly
Dosa
Pongal
```

In [41]:
```
menu[0] = "upma"
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-41-2c50077c4f39> in <module>
----> 1 menu[0] = "upma"

TypeError: 'tuple' object does not support item assignment
```

In [42]:
```
menu = ("upma", "dosa", "pongal")
for m in menu:
    print(m.capitalize())
```

```
Upma
Dosa
Pongal
```

Chapter 5

If Statements

Programming often involves examining a set of conditions and deciding which action to take based on those conditions. Python's if statement allows you to examine the current state of a program and respond appropriately to that state.

syntax:

```
if condition_1:
        statement 1
    elif condition_2:
        statement 2
    else:
        statement 3
```

The Conditional Tests are:

- Checking equality
- Checking Inequality
- Numerical comparisions
- Checking Multiple Conditions
- Checking a value is not in a list
- Boolean Expression

In [1]:
```
car = "subura"
print("Is car is subra I predicted true")
print(car == "subura")
```

```
Is car is subra I predicted true
True
```

In [2]:
```
print("The car is audi I predicted false")
print(car == "audi")
```

```
The car is audi I predicted false
False
```

In [3]:
```
alien_color = ["green", "yellow", "red"]
for ac in alien_color:
    if ac == "yellow":
        print("You have earned 5 points")
    elif ac == "red":
        print("You have earned 10 points")
    else:
        print("GAME OVER")
```

```
GAME OVER
You have earned 5 points
You have earned 10 points
```

In [93]:
```
persons_age = list(range(1,40,2))
for pa in persons_age:
    if pa < 2:
```

```
        print("age: {}: This person is a baby".format(pa))
    elif pa >=2 and pa < 4:
        print("age: {}: This is a toddler".format(pa))
    elif pa >= 4 and pa < 13:
        print("age: {}: This is a Kid".format(pa))
    elif pa >=13 and pa < 20:
        print("age: {}: This is a Teenager".format(pa))
    elif pa >= 20 and pa <65:
        print("age: {}: This is a adult".format(pa))
    else:
        print("age: {}: This is a elder".format(pa))
```

```
age: 1: This person is a baby
age: 3: This is a toddler
age: 5: This is a Kid
age: 7: This is a Kid
age: 9: This is a Kid
age: 11: This is a Kid
age: 13: This is a Teenager
age: 15: This is a Teenager
age: 17: This is a Teenager
age: 19: This is a Teenager
age: 21: This is a adult
age: 23: This is a adult
age: 25: This is a adult
age: 27: This is a adult
age: 29: This is a adult
age: 31: This is a adult
age: 33: This is a adult
age: 35: This is a adult
age: 37: This is a adult
age: 39: This is a adult
```

In [8]:

```
my_fruits = ["apple", "kivi", "orange", "strawberry"]
frd_frunts = ["banana", "apple", "pine"]
for mf in my_fruits:
    if mf in frd_frunts:
        print("You both like {}".format(mf))
    else:
        print("Your frd doesn't like {}".format(mf))
```

```
You both like apple
Your frd doesn't like kivi
Your frd doesn't like orange
Your frd doesn't like strawberry
```

In [1]:

```
users = ["Vision", "Iron man", "Captain america", "Hulk", "Scarlet witch", "Black wi
for u in users:
    if u == "Iron man":
        print("{}: Welcome sir, would you like to see todays top stories.".format(u)
    else:
        print("Welcome {}, Thank you for checking in.".format(u))
```

```
Welcome Vision, Thank you for checking in.
Iron man: Welcome sir, would you like to see todays top stories.
Welcome Captain america, Thank you for checking in.
Welcome Hulk, Thank you for checking in.
Welcome Scarlet witch, Thank you for checking in.
Welcome Black widow, Thank you for checking in.
Welcome Spider man, Thank you for checking in.
Welcome Doctor strange, Thank you for checking in.
Welcome War Machine, Thank you for checking in.
```

In [5]:

```
cuser = ["Vision", "Iron man", "Captain america", "Hulk", "Scarlet witch"]
nuser = ["Black widow", "Spider man", "Doctor strange", "War Machine", "Vision"]
```

```
for nu in nuser:
    if nu in cuser:
        print("{} this username is already taken.".format(nu))
    else:
        print("{} this username is available".format(nu))
```

```
Black widow this username is available
Spider man this username is available
Doctor strange this username is available
War Machine this username is available
Vision this username is already taken.
```

In [7]:

```
lst = list(range(1,10))
for l in lst:
    if l == 1:
        print("{}st".format(l))
    elif l == 2:
        print("{}nd".format(l))
    elif l == 3:
        print("{}rd".format(l))
    else:
        print("{}th".format(l))
```

```
1st
2nd
3rd
4th
5th
6th
7th
8th
9th
```

Chapter 6

Dictionaries

Python's dictionaries, which allow you to connect pieces of related information. You'll learn how to access the information once it's in a dictionary and how to modify that information. Because dictionaries can store an almost limitless amount of information, I'll show you how to loop through the data in a dictionary. Additionally, you'll learn to nest dictionaries inside lists, lists inside dictionaries, and even dictionaries inside other dictionaries.

syntax: `dict = {"key":"value", .....}`

Accessing Values in a Dictionary

To get the value associated with a key, give the name of the dictionary and then place the key inside a set of square brackets, as shown here:

syntax: `func(dict["key"])`

Adding New Key-Value Pairs

Dictionaries are dynamic structures, and you can add new key-value pairs to a dictionary at any time. For example, to add a new key-value pair, you would give the name of the dictionary followed by the new key in square brackets along with the new value.

syntax: `dict["new_key"]="its value"`

Modifying Values in a Dictionary

To modify a value in a dictionary, give the name of the dictionary with the key in square brackets and then the new value you want associated with that key. For example, consider an alien that changes from green to yellow as a game progresses:

syntax: `dict["key already in dict"]="new value"`

Removing Key-Value Pairs

When you no longer need a piece of information that's stored in a dictionary, you can use the del statement to completely remove a key-value pair. All del needs is the name of the dictionary and the key that you want to remove.

syntax: `del dict["key to remove"]`

In [14]:
```
keys = ["first_name", "last_name", "age", "city"]
values = ["tony", "stark", 45, "malibu"]
pdetails = {}
i = 0
while i < len(keys):
    pdetails[keys[i]]=values[i]
    i += 1
print("pdetails = {}".format(pdetails))
```

```
pdetails = {'first_name': 'tony', 'last_name': 'stark', 'age': 45, 'city': 'malibu'}
```

In [16]:
```
fav_num = {"Ironman":25, "CaptainAmerica":28, "SpiderMan":19, "BlackWidow":30, "Doct
for k,v in fav_num.items():
    print("{}'s favarite number is {}".format(k,v))
```

```
Ironman's favarite number is 25
CaptainAmerica's favarite number is 28
SpiderMan's favarite number is 19
BlackWidow's favarite number is 30
DoctorStrange's favarite number is 64
```

Looping through dictionaries

What if you wanted to see everything stored in this user's dictionary? To do so, you could loop through the dictionary using a for loop:

syntax:

`for k,v in dict.items():`

`for k in dict.keys():`

`for v in dict.values():`

In [20]:
```
glossary = {
    'append': 'Adds new element to the list',
    'insert': 'Adds new element to the list at the user specified location',
    'str': 'Used to convert integers to strings',
    'print': 'Prints the user command',
    'del': 'Deletes the given element in the list by the user'
    }
for k,v in glossary.items():
    print("{}: {}\n".format(k,v))
```

```
append: Adds new element to the list

insert: Adds new element to the list at the user specified location

str: Used to convert integers to strings

print: Prints the user command

del: Deletes the given element in the list by the user
```

In [22]:
```
rivers = {"Godavari":"Karnataka", "Ganga":"Uttar Pradesh", "Brahma Putra":"Tibet"}
for k,v in rivers.items():
    print("The {} starts at {}.".format(k.upper(), v.capitalize()))
    print(k)
    print(v)
```

```
The GODAVARI starts at Karnataka.
Godavari
Karnataka
The GANGA starts at Uttar pradesh.
Ganga
Uttar Pradesh
The BRAHMA PUTRA starts at Tibet.
Brahma Putra
Tibet
```

In [24]:
```
favorite_languages = {
    'jen': 'python',
    'sarah': 'c',
    'edward': 'ruby',
    'phil': 'python'
    }
to_poll = ["tony", "captain", "natasha", "sarah", "jen"]

for k in to_poll:
    if k in favorite_languages.keys():
        print("{}: you have already taken the poll.".format(k))
    else:
        print("{} thank you for polling.".format(k))
```

```
tony thank you for polling.
captain thank you for polling.
natasha thank you for polling.
sarah: you have already taken the poll.
jen: you have already taken the poll.
```

In [45]:
```
people = [{"first_name":"tony", "last_name":"stark", "age":45, "city":"malibu"},
        {"first_name":"steve", "last_name":"rogers", "age":120, "city":"brookln"},
        {"first_name":"peter", "last_name":"parker", "age":20, "city":"queens"}]
for p in people:
    pdict = p
    for k,v in pdict.items():
```

```
        print("{} : {}".format(k.capitalize(),v))
    print("\n")
```

```
First_name : tony
Last_name : stark
Age : 45
City : malibu


First_name : steve
Last_name : rogers
Age : 120
City : brookln


First_name : peter
Last_name : parker
Age : 20
City : queens
```

In [56]:

```
pets = []
pets.append({"kind":"dog", "owner":"Tony"})
pets.append({"kind":"cat", "owner":"Peter"})
pets.append({"kind":"bird", "owner":"Vision"})
for p in pets:
    print("{} has a pet {}".format(p["owner"],p["kind"]))
    print("\n")
```

```
Tony has a pet dog


Peter has a pet cat


Vision has a pet bird
```

In [60]:

```
fplaces = {"Tony":["California", "Los Angles", "Sukovia"],"John": ["England", "New Y
for k in fplaces.keys():
    print("{}'s Favorite Places are:".format(k))
    for p in fplaces[k]:
        print(p)
    print("\n")
```

```
Tony's Favorite Places are:
California
Los Angles
Sukovia


John's Favorite Places are:
England
New York
Japan


Maya's Favorite Places are:
Miami
Canada
China
```

In [65]:
```
fnums = {"Tony":list(range(1,10,2)),"John": list(range(5,15,2)), "Maya":list(range(1
for k in fnums.keys():
    print("{}'s Favorite numbers are:".format(k))
    for p in fnums[k]:
        print(p)
    print("\n")
```

```
Tony's Favorite numbers are:
1
3
5
7
9


John's Favorite numbers are:
5
7
9
11
13


Maya's Favorite numbers are:
10
12
14
16
18
```

In [74]:
```
cities={
        'hyderabad':{
            'state':'andhrapradesh',
            'population':'1,25,000',
            'fact':"it was build by nizam's"},
        'chennai':{
            'state':'tamilnadu',
            'population':'2,25,000',
            'fact':"it has world's longest beach"},
        'bombay':{
            'state':'maharastra',
            'population':'3,35,000',
            'fact':"it is the business capital of india"},
        'delhi':{
            'state':'new delhi',
            'population':'1,25,152',
            'fact':'it is the capital of india'}
        }
for k,v in cities.items():
    print("The District {}:".format(k.title()))
    print("\tIt is in {} state".format(v["state"].title()))
    print("\tIt has {} population".format(v["population"]))
    print("\tThe fact about {} is {}".format(k.title(),v["fact"]))
    print("\n")
```

```
The District Hyderabad:
        It is in Andhrapradesh state
        It has 1,25,000 population
        The fact about Hyderabad is it was build by nizam's


The District Chennai:
        It is in Tamilnadu state
        It has 2,25,000 population
```

```
        The fact about Chennai is it has world's longest beach


The District Bombay:
        It is in Maharastra state
        It has 3,35,000 population
        The fact about Bombay is it is the business capital of india


The District Delhi:
        It is in New Delhi state
        It has 1,25,152 population
        The fact about Delhi is it is the capital of india
```

Chapter 7

User inputs and While Loops

User Inputs

The `input()` function takes one argument: the prompt, or instructions, that we want to display to the user so they know what to do. In this example, when Python runs the first line, the user sees the prompt Tell me something, and I will repeat it back to you: . The program waits while the user enters their response and continues after the user presses enter. The response is stored in the variable message, then print message) displays the input back to the user:

syntax: `variable = input(prompt)`

In [78]:
```
cars = ["Honda", "Toyota", "Kia", "Fiat", "Mahindra"]
uinput = input("Which car you want?:")
if uinput.title() in cars:
    print("Glad, we have the {} car.".format(uinput,title()))
else:
    print("Sorry sir we don't have that car.")
```

```
Glad, we have the mahindra car.
```

In [80]:
```
mem = input("How many seats you want?:")
if int(mem) <= 8:
    print("Your table is ready!!!!")
else:
    print("Sorry, you need to wait.")
```

```
Sorry, you need to wait.
```

In [83]:
```
n = int(input("Enter the number:"))
if n % 10 == 0:
    print("It is a multiple of ten.")
else:
    print("It is not multiple of ten!!!!")
```

```
It is a multiple of ten.
```

While loop

The for loop takes a collection of items and executes a block of code once for each item in the collection. In contrast, the while loop runs as long as, or while, a certain condition is true.

syntax: `while (condition):`

In [103…

```
while True:
    top = input("Enter the topping:")
    if top == "quit":
        break
    else:
        print("We will add the {} topping for you.".format(top))
```

```
We will add the chees topping for you.
We will add the macroni topping for you.
We will add the olive topping for you.
```

In [102…

```
while True:
    age = int(input("Enter the age:"))
    if age < 3:
        print("Your ticket is free.")
    elif age >=3 and age < 12:
        print("Your ticket costs $10.")
    else:
        print("Your tictet costs $15.")
```

```
Your ticket is free.
Your ticket costs $10.
Your ticket costs $10.
Your tictet costs $15.
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-102-cda3c8d33d8b> in <module>
      1 while True:
----> 2     age = int(input("Enter the age:"))
      3     if age < 3:
      4         print("Your ticket is free.")
      5     elif age >=3 and age < 12:

ValueError: invalid literal for int() with base 10: 'q'
```

In [110…

```
order_san = ['tuna', 'chicken', 'pastrami', 'mushroom', 'pastrami', 'pork', 'pastram
finished_san = []
while order_san:
    msan = order_san.pop()
    print("I made your {} sanwitch".format(msan))
    finished_san.append(msan)
print("\nfinished_san:{}".format(finished_san))
```

```
I made your turkey sanwitch
I made your beef sanwitch
I made your pastrami sanwitch
I made your pork sanwitch
I made your pastrami sanwitch
I made your mushroom sanwitch
I made your pastrami sanwitch
I made your chicken sanwitch
I made your tuna sanwitch

finished_san:['turkey', 'beef', 'pastrami', 'pork', 'pastrami', 'mushroom', 'pastram
i', 'chicken', 'tuna']
```

In [111…

```
while "pastrami" in finished_san:
    finished_san.remove("pastrami")
```

```
print(finished_san)
```

```
['turkey', 'beef', 'pork', 'mushroom', 'chicken', 'tuna']
```

In [113…

```
responses = {}
ok = True
while ok:
    name = input("Enter your name:")
    place = input("Where would you like to go?:")
    responses[name] = place
    agin = input("Would you like to continue (yes/no):")
    if agin == "no":
        ok = False
    else:
        continue
print(responses)
```

```
{'tony': 'canada', 'con': 'miami'}
```

Chapter 8

Functions

functions, which are named blocks of code that are designed to do one specific job. When you want to perform a particular task that you've defined in a function, you call the name of the function responsible for it. If you need to perform that task multiple times throughout your program, you don't need to type all the code for the same task again and again; you just call the function dedicated to handling that task, and the call tells Python to run the code inside the function. You'll find that using functions makes your programs easier to write, read, test, and fix.

syntax:

```
def func_name(atrribute):

    body of the function

    return


func_name(value)
```

In [1]:

```
def about():
    print("I am learning about functions.")

about()
```

```
I am learning about functions.
```

In [3]:

```
def fav_book(title):
    print("One of my favourite book is {}".format(title.title()))

fav_book("alice in wonderland")
```

```
One of my favourite book is Alice In Wonderland
```

Passing Arguments

Because a function definition can have multiple parameters, a function call may need multiple arguments. You can pass arguments to your functions in a number of ways. You can use positional arguments, which need to be in 136 Chapter 8 the same order the parameters were written; keyword arguments, where each argument consists of a variable name and a value; and lists and dictionaries of values.

Positional Arguments

When you call a function, Python must match each argument in the function call with a parameter in the function definition. The simplest way to do this is based on the order of the arguments provided. Values matched up this way are called `positional arguments`.

syntax:

```
def func_name(arg1,arg2):

    body of funciton

    return


func_name(value1,value2)
```

Keyword Arguments

A keyword argument is a name-value pair that you pass to a function. You directly associate the name and the value within the argument, so when you pass the argument to the function, there's no confusion. Keyword arguments free you from having to worry about correctly ordering your arguments in the function call, and they clarify the role of each value in the function call.

syntax:

```
def func_name(arg1,arg2):

    body of funciton

    return


func_name(arg1 = value1, arg2 = value2)
```

Default Values

When writing a function, you can define a default value for each parameter. If an argument for a parameter is provided in the function call, Python uses the argument value. If not, it uses the parameter's default value. So when you define a default value for a parameter, you can exclude the corresponding argument you'd usually write in the function call. Using default values can simplify your function calls and clarify the ways in which your functions are typically used.

syntax:

```
def func_name(arg1,arg2 = value2):
```

```
    body of funciton

    return
```

```
func_name(value1)
```

In [4]:
```
def make_shirt(size,message):
    print("The {} size and the message to be printed is '{}'".format(size,message.up

# with positinal agruments
make_shirt(32,"I Love Jupyter Notebooks")
```

```
The 32 size and the message to be printed is 'I LOVE JUPYTER NOTEBOOKS'
```

In [5]:
```
make_shirt(size= 34, message="I Love IPython")
```

```
The 34 size and the message to be printed is 'I LOVE IPYTHON'
```

In [8]:
```
def make_shirt2(size=["Large","Medium"],message="I Love Python"):
    for s in size:
        print("The {} size shirt and message is {}".format(s,message.upper()))

make_shirt2()
```

```
The Large size shirt and message is I LOVE PYTHON
The Medium size shirt and message is I LOVE PYTHON
```

In [11]:
```
make_shirt2(size=["free"],message="I Love IPython")
```

```
The free size shirt and message is I LOVE IPYTHON
```

In [12]:
```
def cities(name,contry="India"):
    print("The city {} is in {} country".format(name.title(),contry.title()))

cities("chennai")
```

```
The city Chennai is in India country
```

In [13]:
```
cities(name="newyork",contry="u s a")
```

```
The city Newyork is in U S A country
```

In [14]:
```
cities(name="mumbai")
```

```
The city Mumbai is in India country
```

In [15]:
```
def city_country(city,country):
    city = city.title()
    country = country.title()
    return (city,country)

city_country("chennai","india")
```

Out[15]:
```
('Chennai', 'India')
```

In [2]:
```
def music(artist,album):
```

```
    music = {"artist name":artist, "album":album}
    return music

print(music("michel","Dangerous"))
```

```
{'artist name': 'michel', 'album': 'Dangerous'}
```

In [4]:

```
def music2(artist,album,tracks=""):
    if tracks:
        music = {"artist name":artist, "album":album, "tracks": tracks}
    else:
        music = {"artist name":artist, "album":album}
    return music

print(music2("michel","Dangerous"))
```

```
{'artist name': 'michel', 'album': 'Dangerous'}
```

In [5]:

```
print(music2("michel","dangerous","6"))
```

```
{'artist name': 'michel', 'album': 'dangerous', 'tracks': '6'}
```

In [6]:

```
con = True
while con:
    artist = input("Enter the artist name:")
    album = input("Enter the album:")
    if artist == "q" or album == "q":
        con = False
    else:
        print(music(artist,album))
```

In [7]:

```
def macs(names):
    for n in names:
        print(n.title())

macs(["tony", "strange", "pennywise"])
```

```
Tony
Strange
Pennywise
```

In [8]:

```
def macs2(names):
    for n in names:
        print("Great {}!".format(n.title()))

macs2(["tony", "strange", "pennywise"])
```

```
Great Tony!
Great Strange!
Great Pennywise!
```

In [15]:

```
na = ["tony", "strange", "pennywise"]
gna = []
def unchangedlist(list1,list2):
    while list1:
        l = list1.pop()
        list2.append(l)

def newlist(list2):
    for l in list2:
        print(l.title())
```

```
unchangedlist(na[:],gna)
print(na)
newlist(gna)
```

```
['tony', 'strange', 'pennywise']
Pennywise
Strange
Tony
```

Chapter 9

Classes

Object-oriented programming is one of the most effective approaches to writing software. In object-oriented programming you write classes that represent real-world things and situations, and you create objects based on these classes. When you write a class, you define the general behavior that a whole category of objects can have. When you create individual objects from the class, each object is automatically equipped with the general behavior; you can then give each object whatever unique traits you desire. You'll be amazed how well real-world situations can be modeled with object-oriented programming.

syntax: `class Classname():`

```
        def __init__(self,attribute1,attribute2):
            body

        def method1(self):
            body

        def method2(self):
            body
```

`Classname.method1(attribute1,attribute2)`

In [4]:

```
class Restaurant():
    def __init__(self,name,cusine):
        self.name = name
        self.cusine = cusine

    def desrestaurant(self):
        print("The restaurant {} is a {} cusine".format(self.name.title(),self.cusin

    def restaurantstatus(self):
        print("The restaurant {} is opened now.".format(self.name.title()))

restaurant = Restaurant("rayalaseema","south indian")
restaurant.desrestaurant()
restaurant.restaurantstatus()
```

```
The restaurant Rayalaseema is a South Indian cusine
The restaurant Rayalaseema is opened now.
```

In [6]:

```
rest2 = Restaurant("andhra spice","north indian")
rest3 = Restaurant("food nation", "indian")
rest2.desrestaurant()
```

```
rest2.restaurantstatus()
print("\n")
rest3.desrestaurant()
rest3.restaurantstatus()
```

```
The restaurant Andhra Spice is a North Indian cusine
The restaurant Andhra Spice is opened now.


The restaurant Food Nation is a Indian cusine
The restaurant Food Nation is opened now.
```

In [11]:

```
class User():
    def __init__(self,fname,lname,phone,pin):
        self.fname = fname
        self.lname = lname
        self.phone = phone
        self.pin = pin

    def describe_user(self):
        print("Name: {} {}\nPhone: {}\nPin: {}".format(self.fname.title(),self.lname

    def greet_user(self):
        print("Bounjor!! Mr. {}".format(self.lname.title()))
```

In [12]:

```
user_profile = User("tony", "stark", 58964526548, 245136)
user_profile.describe_user()
user_profile.greet_user()
```

```
Name: Tony Stark
Phone: 58964526548
Pin: 245136
Bounjor!! Mr. Stark
```

In [14]:

```
class Restaurant():
    def __init__(self,name,cusine):
        self.name = name
        self.cusine = cusine
        self.served = 0

    def desrestaurant(self):
        print("The restaurant {} is a {} cusine".format(self.name.title(),self.cusin

    def restaurantstatus(self):
        print("The restaurant {} is opened now.".format(self.name.title()))

    def nofserved(self,serve):
        self.served = serve
        print("The people are served is {}".format(self.served))

    rest = Restaurant("rayalaseema", "south indian")
    rest.desrestaurant()
    rest.restaurantstatus()
```

```
The restaurant Rayalaseema is a South Indian cusine
The restaurant Rayalaseema is opened now.
```

In [16]:

```
rest = Restaurant("rayalaseema", "south indian")
rest.nofserved(10)
```

```
The people are served is 10
```

Chapter 10

Importing extenal Files and Exceptions

Importing and Exporting extenal file types like .txt .json

To work with files and save data will make your programs easier for people to use. Users will be able to choose what data to enter and when to enter it. People can run your program, do some work, and then close the program and pick up where they left off later. Learning to handle exceptions will help you deal with situations in which files don't exist and deal with other problems that can cause your programs to crash. This will make your programs more robust when they encounter bad data, whether 190 Chapter 10 it comes from innocent mistakes or from malicious attempts to break your programs. With the skills you'll learn in this chapter, you'll make your pro□grams more applicable, usable, and stable.

Reading from a files

syntax:

```
with open ("path to file to open") as dummyname:

            variable = dummyname.read()   to read the entire file

            variable = dummyname.readlines()    to read the each line
```

In [19]:
```
with open ("D:\Python\Phase 1\Projects and Exerises\Resources\learning_python.txt")
    lines = lp.read()

print(lines)
```

```
In python you can print with single line.
In python you can create AI.
In python you can write coding for games.
In python you can do machine learning.
```

In [25]:
```
with open ("D:\Python\Phase 1\Projects and Exerises\Resources\learning_python.txt")
    for l in lp:
        print(l)
```

```
In python you can print with single line.

In python you can create AI.

In python you can write coding for games.

In python you can do machine learning.
```

In [27]:
```
with open ("D:\Python\Phase 1\Projects and Exerises\Resources\learning_python.txt")
    lines = lp.readlines()
for l in lines:
    print(l)
```

```
In python you can print with single line.

In python you can create AI.
```

```
In python you can write coding for games.

In python you can do machine learning.
```

Writing onto a file

syntax:

```
with open("Path to file", "w") as dummyname:

            dummyname.write("lines to write onto the file")
```

- to read use 'r'
- to write use 'w'
- to append use 'a'
- to use read an write use 'r+'

In [32]:
```
with open ("D:\Python\Phase 1\Projects and Exerises\Resources\pynb.txt", "a") as tow

    while True:
        ip = input("Enter your name(q to exit): ")
        if ip == "q":
            break
        else:
            towrite.write("{}\n".format(ip.title()))
```

Exceptions

Python uses special objects called exceptions to manage errors that arise during a program's execution. Whenever an error occurs that makes Python unsure what to do next, it creates an exception object. If you write code that handles the exception, the program will continue running. If you don't handle the exception, the program will halt and show a traceback, which includes a report of the exception that was raised.

Exceptions are handled with try-except blocks. A try-except block asks Python to do something, but it also tells Python what to do if an exception is raised. When you use try-except blocks, your programs will continue running even if things start to go wrong. Instead of tracebacks, which can be confusing for users to read, users will see friendly error messages that you write.

Zero Division Error

In [33]:
```
print("Give me two numbers, and I'll divide them.")
print("Enter 'q' to quit.")

while True:
    first_number = input("\nFirst number: ")
    if first_number == 'q':
        break
    second_number = input("Second number: ")
    try:
        answer = int(first_number) / int(second_number)
    except ZeroDivisionError:
        print("You can't divide by 0!")
    else:
        print(answer)
```

```
Give me two numbers, and I'll divide them.
Enter 'q' to quit.
You can't divide by 0!
0.2
0.5
You can't divide by 0!
You can't divide by 0!
You can't divide by 0!
```

File Not Found Error

In [35]:

```
filename = 'alice.txt'

try:
    with open(filename, encoding='utf-8') as f_obj:
        contents = f_obj.read()
except FileNotFoundError as e:
    msg = "Sorry, the file " + filename + " does not exist."
    print(msg)
else:
    # Count the approximate number of words in the file.
    words = contents.split()
    num_words = len(words)
    print("The file {} has about {} words.".format(filename,num_words))
```

```
The file alice.txt has about 29461 words.
```

In [36]:

```
filename = 'alice.txt'

try:
    with open(filename, encoding='utf-8') as f_obj:
        contents = f_obj.read()
except FileNotFoundError as e:
    msg = "Sorry, the file " + filename + " does not exist."
    print(msg)
else:
    # Count the approximate number of words in the file.
    words = contents.split()
    num_words = len(words)
    print("The file {} has about {} words.".format(filename,num_words))
```

```
Sorry, the file alice.txt does not exist.
```

Part II -- Data Analysis and Data Visualization

Data visualization involves exploring data through visual representations. It's closely associated with data mining, which uses code to explore the patterns and connections in a data set. A data set can be just a small list of numbers that fits in one line of code or many gigabytes of data. Making beautiful representations of data is about more than pretty pictures. When you have a simple,visually appealing representation of a data set, its meaning becomes clear to viewers. People will see patterns and significance in your data sets that they never knew existed.

When I say "data", what am I referring to exactly? The primary focus is on structured data, a deliberately vague term that encompasses many different common forms of data, such as

- Multidimensional arrays (matrices)
- Tabular or spreadsheet-like data in which each column may be a different type (string, numeric, date, or otherwise). This includes most kinds of data commonly stored in relational databases or tab- or comma-delimited text files

- Multiple tables of data interrelated by key columns (what would be primary or foreign keys for a SQL user)
- Evenly or unevenly spaced time series

Essential Python Libraries

Numpy

NumPy, short for Numerical Python, is the foundational package for scientific com□puting in Python. The majority of this book will be based on NumPy and libraries built on top of NumPy.

It provides, among other things:

- A fast and efficient multidimensional array object ndarray
- Functions for performing element-wise computations with arrays or mathematical operations between arrays
- Tools for reading and writing array-based data sets to disk
- Linear algebra operations, Fourier transform, and random number generation
- Tools for integrating connecting C, C++, and Fortran code to Python

Pandas

Pandas provides rich data structures and functions designed to make working with structured data fast, easy, and expressive. It is,as you will see, one of the critical in□gredients enabling Python to be a powerful and productive data analysis environment. The primary object in pandas that will be used in this book is the DataFrame, a two□dimensional tabular, column-oriented data structure with both row and column labels. pandas combines the high performance array-computing features of NumPy with the flexible data manipulation capabilities of spreadsheets and relational databases (such as SQL). It provides sophisticated indexing functionality to make it easy to reshape, slice and dice, perform aggregations, and select subsets of data.

The pandas name itself is derived from panel data, an econometrics term for multidimensional structured data sets, and Python data analysis itself

Matplotlib

matplotlib is the most popular Python library for producing plots and other 2D data visualizations. It was originally created by John D. Hunter (JDH) and is now maintained by a large team of developers. It is well-suited for creating plots suitable for publication. It integrates well with IPython (see below), thus providing a comfortable interactive environment for plotting and exploring data. The plots are also interactive; you can zoom in on a section of the plot and pan around the plot using the toolbar in the plot window.

Chapter 11

Numpy Library

Importing numpy library

```
import numpy as np
```

```
from numpy import *
```

In [1]:
```
import numpy as np
```

Numpy ndarry: Multidimensional array

Creating array with python lists

In [5]:
```
data = list(range(1,10))
arr = np.array(data)
print("{}\n{}".format(arr,arr.shape))
```

```
[1 2 3 4 5 6 7 8 9]
(9,)
```

In [11]:
```
data2 = [[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]]
arr2 = np.array(data2)
print("{}\n{}".format(arr2,arr2.shape))
```

```
[[ 1  2  3  4  5]
 [ 6  7  8  9 10]]
(2, 5)
```

In [12]:
```
my_matrix = [[1,2,3],[4,5,6],[7,8,9]]
my_matrix
```

Out[12]:
```
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

In [13]:
```
np.array(my_matrix)
```

Out[13]:
```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

Built-in Methods

arange

`arange` is an array-valued version of the built-in Python range function:

Return evenly spaced values within a given interval

syntax: `np.arange(start,end,step)`

In [14]:
```
np.arange(0,10)
```

Out[14]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

In [15]:
```
np.arange(0,11,2)
```

Out[15]:
```
array([ 0,  2,  4,  6,  8, 10])
```

zeros and ones

Generate arrays of zeros and ones

In [49]:
```
np.zeros((5,5))
```

Out[49]:
```
array([[0., 0., 0., 0., 0.],
```

```
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.]])
```

In [17]:
```
np.ones((5,5))
```

Out[17]:
```
array([[1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.]])
```

linspace

Return evely spaced numbers over a specified interval

`np.linspace( start, stop, num=50, endpoint=True,retstep=False, dtype=None, axis=0, )`

Parameters

start : array_like The starting value of the sequence.

stop : array_like The end value of the sequence, unless endpoint is set to False. In that case, the sequence consists of all but the last of num + 1 evenly spaced samples, so that stop is excluded. Note that the step size changes when endpoint is False.

num : int, optional Number of samples to generate. Default is 50. Must be non-negative.

endpoint : bool, optional If True, stop is the last sample. Otherwise, it is not included. Default is True.

retstep : bool, optional If True, return (samples, step), where step is the spacing between samples.

dtype : dtype, optional The type of the output array. If dtype is not given, the data type is inferred from start and stop. The inferred dtype will never be an integer; float is chosen even if the arguments would produce an array of integers.

axis : int, optional The axis in the result to store the samples. Relevant only if start or stop are array-like. By default (0), the samples will be along a new axis inserted at the beginning. Use -1 to get an axis at the end.

In [20]:
```
np.linspace(0,10)  #num = 50
```

Out[20]:
```
array([ 0.        ,  0.20408163,  0.40816327,  0.6122449 ,  0.81632653,
        1.02040816,  1.2244898 ,  1.42857143,  1.63265306,  1.83673469,
        2.04081633,  2.24489796,  2.44897959,  2.65306122,  2.85714286,
        3.06122449,  3.26530612,  3.46938776,  3.67346939,  3.87755102,
        4.08163265,  4.28571429,  4.48979592,  4.69387755,  4.89795918,
        5.10204082,  5.30612245,  5.51020408,  5.71428571,  5.91836735,
        6.12244898,  6.32653061,  6.53061224,  6.73469388,  6.93877551,
        7.14285714,  7.34693878,  7.55102041,  7.75510204,  7.95918367,
        8.16326531,  8.36734694,  8.57142857,  8.7755102 ,  8.97959184,
        9.18367347,  9.3877551 ,  9.59183673,  9.79591837, 10.        ])
```

In [21]:
```
np.linspace(0,10,3)  # num = 3
```

Out[21]:
```
array([ 0.,  5., 10.])
```

eye

Creates an identity matrix

`np.eye(N, M=None, k=0, dtype=<class 'float'>, order='C', *, like=None)`

Parameters

N : int Number of rows in the output.

M : int, optional Number of columns in the output. If None, defaults to `N`.

k : int, optional Index of the diagonal: 0 (the default) refers to the main diagonal, a positive value refers to an upper diagonal, and a negative value to a lower diagonal.

dtype : data-type, optional Data-type of the returned array.

order : {'C', 'F'}, optional Whether the output should be stored in row-major (C-style) or column-major (Fortran-style) order in memory.

like : array_like Reference object to allow the creation of arrays which are not NumPy arrays. If an array-like passed in as `like` supports the `__array_function__` protocol, the result will be defined by it. In this case, it ensures the creation of an array object compatible with that passed in via this argument. .. note:: The `like` keyword is an experimental feature pending on acceptance of :ref: `NEP 35 <NEP35>`.

In [22]:
```
np.eye(4)
```

Out[22]:
```
array([[1., 0., 0., 0.],
       [0., 1., 0., 0.],
       [0., 0., 1., 0.],
       [0., 0., 0., 1.]])
```

In [32]:
```
np.eye(4,4,-1)
```

Out[32]:
```
array([[0., 0., 0., 0.],
       [1., 0., 0., 0.],
       [0., 1., 0., 0.],
       [0., 0., 1., 0.]])
```

In [33]:
```
np.eye(4,4,+1)
```

Out[33]:
```
array([[0., 1., 0., 0.],
       [0., 0., 1., 0.],
       [0., 0., 0., 1.],
       [0., 0., 0., 0.]])
```

numpy other functions

array: Convert input data (list, tuple, array, or other sequence type) to an ndarray either by inferring a dtype or explicitly specifying a dtype. Copies the input data by default.

asarray: Convert input to ndarray, but do not copy if the input is already an ndarray

arange: Like the built-in range but returns an ndarray instead of a list.

ones, ones_like: Produce an array of all 1's with the given shape and dtype. ones_like takes another array and produces a ones array of the same shape and dtype.

zeros, zeros_like: Like ones and ones_like but producing arrays of 0's instead 82 | Chapter 4: NumPy Basics: Arrays and Vectorized Computation

empty, empty_like: Create new arrays by allocating new memory, but do not populate with any values like ones and zeros

eye, identity: Create a square N x N identity matrix (1's on the diagonal and 0's elsewhere)

Generating random numbers array

```
np.random.randint(low, high=None, size=None, dtype=int)
```

Return random integers from low (inclusive) to high (exclusive).

Return random integers from the "discrete uniform" distribution of the specified dtype in the "half-open" interval [low, high). If high is None (the default), then results are from [0, low).

Parameters

low : int or array-like of ints Lowest (signed) integers to be drawn from the distribution (unless high=None, in which case this parameter is one above the *highest* such integer).

high : int or array-like of ints, optional If provided, one above the largest (signed) integer to be drawn from the distribution (see above for behavior if high=None). If array-like, must contain integer values

size : int or tuple of ints, optional Output shape. If the given shape is, e.g., (m, n, k), then m *n* k samples are drawn. Default is None, in which case a single value is returned.

dtype : dtype, optional Desired dtype of the result. Byteorder must be native. The default value is int.

In [41]:
```
np.random.randint(0,50,(3,3))
```

Out[41]:
```
array([[20, 29, 33],
       [10, 27, 37],
       [38, 27, 29]])
```

```
np.random.rand(d0, d1, ..., dn)
```

Random values in a given shape.

Create an array of the given shape and populate it with random samples from a uniform distribution over [0,1).

Parameters

d0, d1, ..., dn : int, optional The dimensions of the returned array, must be non-negative. If no argument is given a single Python float is returned.

In [46]:
```
np.random.rand(3,3,5)
```

```
array([[[0.89100697, 0.11199101, 0.77069665, 0.27744243, 0.6479809 ],
```

Out[46]:
```
        [0.18868817, 0.53621358, 0.30614024, 0.97378593, 0.4913553 ],
        [0.17906539, 0.58048403, 0.19314575, 0.94362145, 0.52491366]],

       [[0.27867623, 0.39428251, 0.42541797, 0.62257382, 0.50397307],
        [0.62879052, 0.36727558, 0.43287833, 0.19377203, 0.74067685],
        [0.90803812, 0.92776492, 0.7517891 , 0.81735307, 0.65963233]],

       [[0.76047731, 0.81884497, 0.06068867, 0.07991164, 0.71598997],
        [0.25942675, 0.1231444 , 0.15641442, 0.20003497, 0.39489586],
        [0.22046615, 0.7237785 , 0.81382166, 0.62687922, 0.49894853]]])
```

`np.random.randn(d0, d1, ..., dn)`

Return a sample (or samples) from the "standard normal" distribution.

If positive int_like arguments are provided, randn generates an array of shape (d0, d1, ..., dn), filled with random floats sampled from a univariate "normal" (Gaussian) distribution of mean 0 and variance 1. A single float randomly sampled from the distribution is returned if no argument is provided.

Parameters

d0, d1, ..., dn : int, optional The dimensions of the returned array, must be non-negative. If no argument is given a single Python float is returned.

In [50]:
```
np.random.randn(3,3,4)
```

Out[50]:
```
array([[[ 1.49761254, -0.73242414, -0.23701634,  1.27659329],
        [-1.10305294, -0.03930065,  1.65367143,  1.75852865],
        [ 0.53252265, -0.33383566,  0.92404222,  1.20355084]],

       [[ 0.35143721, -0.49636959, -1.81647021, -1.33545712],
        [ 0.52276633,  1.93350776,  0.79403935, -1.04991672],
        [ 0.03698526,  1.43675821,  0.34513104,  0.9469469 ]],

       [[-0.30748981, -0.0281714 ,  0.05708983,  0.3574494 ],
        [ 0.63617781,  0.46543418,  1.51706091,  0.94377665],
        [ 0.5810443 , -1.14374854, -0.56521072,  0.24252177]]])
```

Some more numpy.random functions

In [3]:
```
from IPython.display import Image, display
```

In [140…
```
display(Image(filename="D:\\Python\\Python handbook\\5.jpg"))
```

Table 4-8. Partial list of numpy.random functions

Function	Description
seed	Seed the random number generator
permutation	Return a random permutation of a sequence, or return a permuted range
shuffle	Randomly permute a sequence in place
rand	Draw samples from a uniform distribution
randint	Draw random integers from a given low-to-high range
randn	Draw samples from a normal distribution with mean 0 and standard deviation 1 (MATLAB-like interface)
binomial	Draw samples a binomial distribution
normal	Draw samples from a normal (Gaussian) distribution
beta	Draw samples from a beta distribution
chisquare	Draw samples from a chi-square distribution
gamma	Draw samples from a gamma distribution
uniform	Draw samples from a uniform [0, 1) distribution

Operation between Array and Scalar

Arrays are important because they enable you to express batch operations on data without writing any for loops. This is usually called vectorization. Any arithmetic operations between equal-size arrays applies the operation elementwise:

`array1 + array2` | `array1 - array2`

`array1 * array2` | `array1 / array2`

In [62]:
```
arr1 = np.random.randint(0,50,10).reshape(2,5)
arr2 = np.random.randint(50,100,10).reshape(2,5)
```

Out[62]:
```
array([[ 1, 30,  7,  1, 20],
       [34, 14, 17, 48, 48]])
```

In [72]:
```
print("sum: {},\n\nsub: {},\n\nMul: {},\n\nDiv: {}".format((arr1+arr2),(arr1-arr2),(
```

```
sum: [[ 91  92  86  89 100]
 [100  81 116 139 125]],

sub: [[-89 -32 -72 -87 -60]
 [-32 -53 -82 -43 -29]],

Mul: [[  90 1860  553   88 1600]
 [2244  938 1683 4368 3696]],

Div: [[0.01111111 0.48387097 0.08860759 0.01136364 0.25      ]
 [0.51515152 0.20895522 0.17171717 0.52747253 0.62337662]]
```

Arithmetic operations with scalars are as you would expect, propagating the value to each element:

`1/array` | `array * integer`

`array ** integer`

In [73]:
```
1/arr1
```

Out[73]:
```
array([[1.        , 0.03333333, 0.14285714, 1.        , 0.05      ],
       [0.02941176, 0.07142857, 0.05882353, 0.02083333, 0.02083333]])
```

In [74]:
```
arr1*2
```

Out[74]:
```
array([[ 2, 60, 14,  2, 40],
       [68, 28, 34, 96, 96]])
```

In [75]:
```
arr1**2
```

Out[75]:
```
array([[   1,  900,   49,    1,  400],
       [1156,  196,  289, 2304, 2304]], dtype=int32)
```

Basic Indexing and Slicing

NumPy array indexing is a rich topic, as there are many ways you may want to select a subset of your data or individual elements. One-dimensional arrays are simple; `on the surface they act similarly to Python lists:`

In [76]:
```
arr = np.arange(0,10)
arr
```

Out[76]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

In [78]:
```
arr[5]          #array[index]
```

Out[78]:
```
5
```

In [79]:
```
arr[5:8]       #array[start index:end index]
```

Out[79]:
```
array([5, 6, 7])
```

broadcast_arrays

As you can see, if you assign a scalar value to a slice, as in `arr[5:8] = 100`*, the value is propagated (or broadcasted henceforth) to the entire selection. An important first distinction from lists is that array slices are views on the original array. This means that the data is not copied, and any modifications to the view will be reflected in the source array:*

In [104…
```
arr[5:8] = 100
arr
```

Out[104…
```
array([  0,   1,   2,   3,   4, 100, 100, 100,   8,   9])
```

With higher dimensional arrays, you have many more options. In a two-dimensional array, the elements at each index are no longer scalars but rather one-dimensional arrays:

`array[row][column]`

In [80]:
```
arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
arr2d
```

Out[80]:
```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

In [86]:
```
arr2d[0]
```

Out[86]:
```
array([1, 2, 3])
```

In [88]:
```
arr2d[1][2]
```

Out[88]:
```
6
```

In [89]:
```
arr2d[1,2]
```

Out[89]:
```
6
```

In multidimensional arrays, if you omit later indices, the returned object will be a lower‐dimensional ndarray consisting of all the data along the higher dimensions.

In [91]:
```
arr3d = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
arr3d
```

Out[91]:
```
array([[[ 1,  2,  3],
        [ 4,  5,  6]],

       [[ 7,  8,  9],
        [10, 11, 12]]])
```

In [92]:
```
arr3d[0]
```

Out[92]:
```
array([[1, 2, 3],
       [4, 5, 6]])
```

In [93]:
```
arr3d[1,0]
```

Out[93]:
```
array([7, 8, 9])
```

indexing with slice

In [94]:
```
arr2d[:2]
```

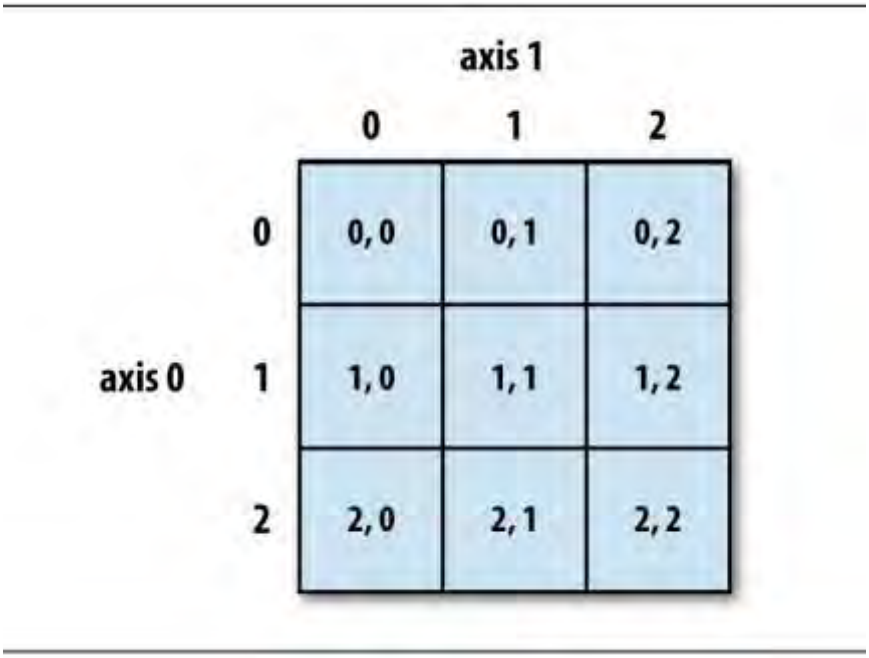
Out[94]:
```
array([[1, 2, 3],
       [4, 5, 6]])
```

In [96]:
```
arr2d[:2,1:]
```

Out[96]:
```
array([[2, 3],
       [5, 6]])
```

numpyarray index

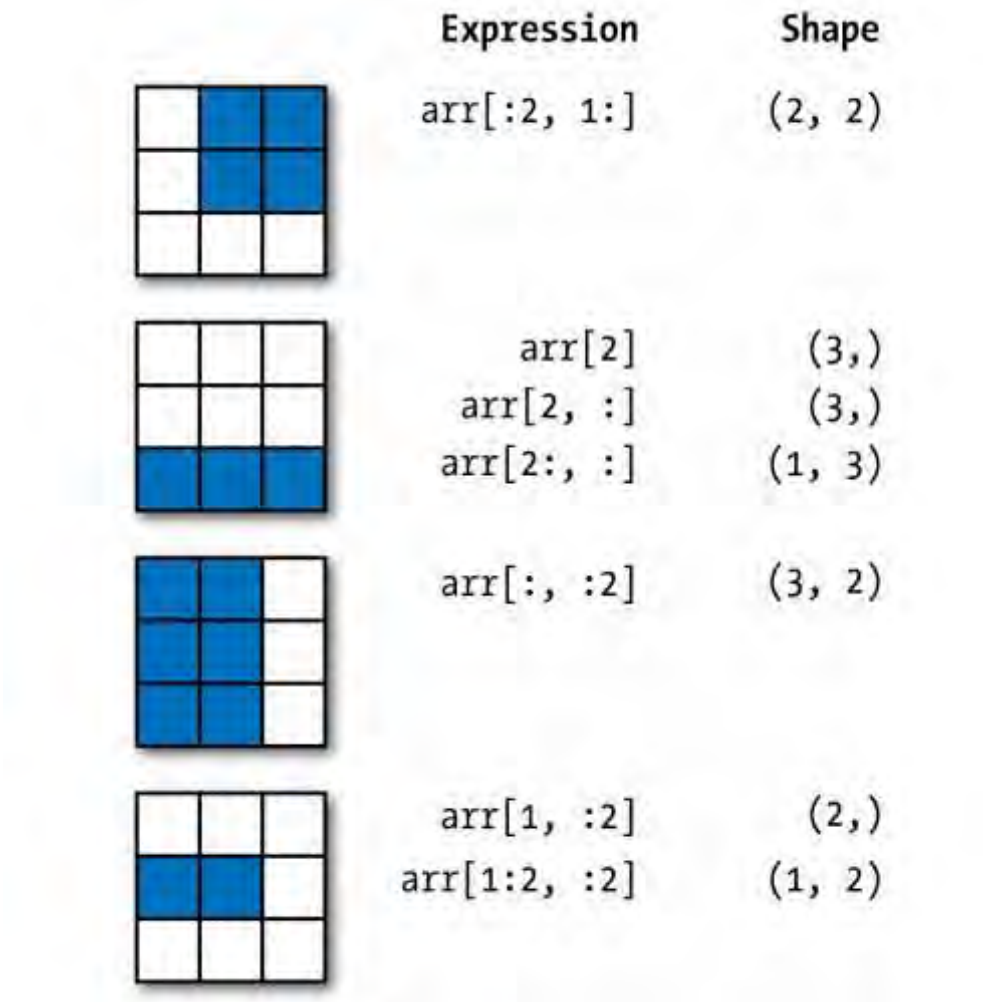
In [102…
```
display(Image(filename="D:\\Python\\Python handbook\\2.jpg"))
```



Slicing the 2d array

In [103…

```
display(Image(filename="D:\\Python\\Python handbook\\1.jpg"))
```



Boolean Indexing

In [109…

```
names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'])
names
```

Out[109…
```
array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'], dtype='<U4')
```

In [110…
```
data = np.random.randn(7,4)
data
```

Out[110…
```
array([[ 0.67057691,  1.45176647,  0.40819274, -0.50685504],
       [ 0.6470143 , -0.56615321,  0.63482171, -0.90954325],
       [ 1.58313901, -0.13867972, -1.33297867, -0.48150745],
       [ 0.57602293,  1.53587786,  0.27754898,  0.43445956],
       [ 0.1520492 , -1.27072812, -0.96655937, -0.16340589],
       [-0.21152496, -0.61604685, -1.17486001, -0.95609148],
       [ 0.51441198, -1.66782702,  0.49865587, -2.50273575]])
```

In [111…
```
names == "Bob"
```

Out[111…
```
array([ True, False, False,  True, False, False, False])
```

In [112…
```
data[names == "Bob"]
```

Out[112…
```
array([[ 0.67057691,  1.45176647,  0.40819274, -0.50685504],
       [ 0.57602293,  1.53587786,  0.27754898,  0.43445956]])
```

In [115…
```
data[data < 0] = 0
data
```

Out[115…
```
array([[0.67057691, 1.45176647, 0.40819274, 0.        ],
       [0.6470143 , 0.        , 0.63482171, 0.        ],
       [1.58313901, 0.        , 0.        , 0.        ],
       [0.57602293, 1.53587786, 0.27754898, 0.43445956],
       [0.1520492 , 0.        , 0.        , 0.        ],
       [0.        , 0.        , 0.        , 0.        ],
       [0.51441198, 0.        , 0.49865587, 0.        ]])
```

Trasforming array and swaping Axes

`array.reshape(tuple)`

tuple (no of rows,no of columns)

In [118…
```
arr3d
```

Out[118…
```
array([[[ 1,  2,  3],
        [ 4,  5,  6]],

       [[ 7,  8,  9],
        [10, 11, 12]]])
```

In [121…
```
arr3d.reshape(3,4)
```

Out[121…
```
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
```

In [123…
```
arr3d.T
```

Out[123…
```
array([[[ 1,  7],
        [ 4, 10]],

       [[ 2,  8],
```

```
       [ 5, 11]],

      [[ 3,  9],
       [ 6, 12]]])
```

In [129…
```
arr3d.swapaxes(1,2)
```

Out[129…
```
array([[[ 1,  4],
        [ 2,  5],
        [ 3,  6]],

       [[ 7, 10],
        [ 8, 11],
        [ 9, 12]]])
```

Universal Functions

In [130…
```
display(Image(filename="D:\\Python\\Python handbook\\3.jpg"))
```

Function	Description
abs, fabs	Compute the absolute value element-wise for integer, floating point, or complex values. Use fabs as a faster alternative for non-complex-valued data
sqrt	Compute the square root of each element. Equivalent to arr ** 0.5
square	Compute the square of each element. Equivalent to arr ** 2
exp	Compute the exponent e^x of each element
log, log10, log2, log1p	Natural logarithm (base e), log base 10, log base 2, and $\log(1+x)$, respectively
sign	Compute the sign of each element: 1 (positive), 0 (zero), or -1 (negative)
ceil	Compute the ceiling of each element, i.e. the smallest integer greater than or equal to each element
floor	Compute the floor of each element, i.e. the largest integer less than or equal to each element
rint	Round elements to the nearest integer, preserving the dtype
modf	Return fractional and integral parts of array as separate array
isnan	Return boolean array indicating whether each value is NaN (Not a Number)
isfinite, isinf	Return boolean array indicating whether each element is finite (non-inf, non-NaN) or infinite, respectively
cos, cosh, sin, sinh, tan, tanh	Regular and hyperbolic trigonometric functions
arccos, arccosh, arcsin, arcsinh, arctan, arctanh	Inverse trigonometric functions
logical_not	Compute truth value of not x element-wise. Equivalent to -arr.

In [132…
```
np.sqrt(arr2d)
```

Out[132…
```
array([[1.        , 1.41421356, 1.73205081],
       [2.        , 2.23606798, 2.44948974],
       [2.64575131, 2.82842712, 3.        ]])
```

In [131…
```
display(Image(filename="D:\\Python\\Python handbook\\4.jpg"))
```

Function	Description
add	Add corresponding elements in arrays
subtract	Subtract elements in second array from first array
multiply	Multiply array elements
divide, floor_divide	Divide or floor divide (truncating the remainder)
power	Raise elements in first array to powers indicated in second array
maximum, fmax	Element-wise maximum. fmax ignores NaN
minimum, fmin	Element-wise minimum. fmin ignores NaN
mod	Element-wise modulus (remainder of division)
copysign	Copy sign of values in second argument to values in first argument
greater, greater_equal, less, less_equal, equal, not_equal	Perform element-wise comparison, yielding boolean array. Equivalent to infix operators >, >=, <, <=, ==, !=
logical_and, logical_or, logical_xor	Compute element-wise truth value of logical operation. Equivalent to infix operators &, |, ^

In [139…
```
np.max(arr2d)
```

Out[139… 9

Chapter 12

Pandas Library

It contains high-level data structures and manipulation tools designed to make data analysis fast and easy in Python. pandas is built on top of NumPy and makes it easy to use in NumPy-centric applications.

- Data structures with labeled axes supporting automatic or explicit data alignment. This prevents common errors resulting from misaligned data and working with differently-indexed data coming from different sources.
- Integrated time series functionality.
- The same data structures handle both time series data and non-time series data.
- Arithmetic operations and reductions (like summing across an axis) would pass on the metadata (axis labels).
- Flexible handling of missing data.
- Merge and other relational operations found in popular database databases (SQL□based, for example).

Importing Pandas Library

In [2]:
```
import pandas as pd
```

To get started with pandas, you will need to get comfortable with its two workhorse data structures:

- Series

- DataFrame.

Series

A Series is a one-dimensional array-like object containing an array of data (of any NumPy data type) and an associated array of data labels, called its index.

`pd.Series(data: [ndarray | Dict[_str, ndarray] | Sequence], index: [str | int | Series | List ], dtype=...)`

In [3]:
```
obj = pd.Series([4, 7, -5, 3])
obj
```

Out[3]:
```
0    4
1    7
2   -5
3    3
dtype: int64
```

In [4]:
```
obj.values
```

Out[4]:
```
array([ 4,  7, -5,  3], dtype=int64)
```

In [5]:
```
obj.index
```

Out[5]:
```
RangeIndex(start=0, stop=4, step=1)
```

In [11]:
```
obj = pd.Series([4, 7, -5, 3], index=["a", "b", "e", "g"])
obj
```

Out[11]:
```
a    4
b    7
e   -5
g    3
dtype: int64
```

In [10]:
```
sdata = {'Ohio': 35000, 'Texas': 71000, 'Oregon': 16000, 'Utah': 5000}
obj2 = pd.Series(sdata)
obj2
```

Out[10]:
```
Ohio      35000
Texas     71000
Oregon    16000
Utah       5000
dtype: int64
```

Acessing data from the Series

`seriesname["index"]`

In [10]:
```
obj2["Ohio"]
```

Out[10]:
```
35000
```

In [11]:
```
obj[2]
```

Out[11]:
```
-5
```

Changing the values in series

```
seriesname["index"] = value
```

In [13]:
```
obj[2] = 6
obj[2]
```

Out[13]:
```
6
```

Arithmetic operations

To perform these operations the index must be same or mostly like

In [9]:
```
obj3 =pd.Series({"Ohio": 12000, "Texas": 25000, "Oregon": 9600, "Utah": 7820, "Cali"
obj3
```

Out[9]:
```
Ohio      12000
Texas     25000
Oregon     9600
Utah       7820
Cali       8900
dtype: int64
```

In [16]:
```
obj2
```

Out[16]:
```
Ohio      35000
Texas     71000
Oregon    16000
Utah       5000
dtype: int64
```

In [24]:
```
objs = obj2 + obj3
objs
```

Out[24]:
```
Cali          NaN
Ohio      47000.0
Oregon    25600.0
Texas     96000.0
Utah      12820.0
dtype: float64
```

In [26]:
```
objsb = obj2 - obj3
objsb
```

Out[26]:
```
Cali          NaN
Ohio      23000.0
Oregon     6400.0
Texas     46000.0
Utah      -2820.0
dtype: float64
```

In [28]:
```
objm = obj2 * obj3
objm
```

Out[28]:
```
Cali              NaN
Ohio      4.200000e+08
Oregon    1.536000e+08
Texas     1.775000e+09
Utah      3.910000e+07
dtype: float64
```

In [29]:
```
objd = obj2 / obj3
objd
```

Out[29]:
```
Cali          NaN
Ohio      2.916667
Oregon    1.666667
Texas     2.840000
Utah      0.639386
dtype: float64
```

In [37]:
```
obj2["Ohio"] + obj3["Cali"]
```

Out[37]:
```
43900
```

The data thats not find in other series will be treated as not available | NaN | Missing

The `isnull` *and* `notnull` *functions in pandas should be used to detect missing data:*

In [30]:
```
pd.isnull(objs)
```

Out[30]:
```
Cali       True
Ohio      False
Oregon    False
Texas     False
Utah      False
dtype: bool
```

In [31]:
```
objs.isnull()
```

Out[31]:
```
Cali       True
Ohio      False
Oregon    False
Texas     False
Utah      False
dtype: bool
```

In [34]:
```
objs.isna()
```

Out[34]:
```
Cali       True
Ohio      False
Oregon    False
Texas     False
Utah      False
dtype: bool
```

In [32]:
```
objs.notna()
```

Out[32]:
```
Cali      False
Ohio       True
Oregon     True
Texas      True
Utah       True
dtype: bool
```

In [33]:
```
objs.notnull()
```

Out[33]:
```
Cali      False
Ohio       True
Oregon     True
```

```
Texas      True
Utah       True
dtype: bool
```

Data alignment features are addressed as a separate topic.

Both the Series object itself and its index have a name attribute, which integrates with other key areas of pandas functionality:

For values: `Seriesname.name = "name"`

For index: `Seriesname.index.name = "name"`

In [35]:
```
obj3.name = "Population"
obj3.index.name = "state"
obj3
```

Out[35]:
```
state
Ohio      12000
Texas     25000
Oregon     9600
Utah       7820
Cali       8900
Name: Population, dtype: int64
```

DataFrames

A DataFrame represents a tabular, spreadsheet-like data structure containing an ordered collection of columns, each of which can be a different value type (numeric, string, boolean, etc.). `The DataFrame has both a row and column index;` it can be thought of as a dict of Series (one for all sharing the same index). Compared with other such DataFrame-like structures you may have used before (like R's data.frame), row-oriented and column-oriented operations in DataFrame are treated roughly symmetrically. Under the hood, the data is stored as one or more two-dimensional blocks rather than a list, dict, or some other collection of one-dimensional arrays.

pd.DataFrame(`data:` ndarray | List | Dict | Sequence | DataFrame, `index:` Index | ndarray | List | Dict | Sequence | Series, `columns:` ndarray | List | Dict | Sequence, `dtype=`)

In [74]:
```
df = pd.DataFrame(data= np.random.randn(5,4),index= 'A B C D E'.split(),columns= 'W
df
```

Out[74]:

	W	X	Y	Z
A	0.783226	-0.260299	-0.258958	0.042018
B	-1.190230	0.875860	0.433010	-1.240584
C	-0.479875	1.013206	-0.462740	-0.981134
D	0.399906	1.382583	-0.682202	0.175025
E	0.153638	-0.229203	-0.439973	0.681553

A column in a DataFrame can be retrieved as a Series either by dict-like notation or by attribute:

In [5]:
```
df["W"]
```

Out[5]:
```
A   -0.771294
B   -0.284151
C    0.840934
```

```
D   -1.503204
E   -0.505033
Name: W, dtype: float64
```

Columns can be modified by assignment. For example, the empty 'new' column could be assigned a sum of value or an array of values:

In [16]:
```
df['new'] = df['W'] + df['Y']
df
```

Out[16]:

	W	X	Y	Z	new
A	1.849800	1.387938	-1.137278	-0.123643	0.712522
B	0.635781	-0.213673	0.444087	-1.345951	1.079867
C	-1.188844	0.822141	-0.043321	1.888114	-1.232164
D	-0.871444	1.767228	0.566999	0.491481	-0.304445
E	0.249339	-1.767751	0.577606	-0.269734	0.826946

When assigning lists or arrays to a column, the value's length must match the length of the DataFrame. If you assign a Series, it will be instead conformed exactly to the DataFrame's index, inserting missing values in any holes

In [7]:
```
del df['new']
df.columns
```

Out[7]:
```
Index(['W', 'X', 'Y', 'Z'], dtype='object')
```

The column returned when indexing a DataFrame is a view on the underlying data, not a copy. Thus, any in-place modifications to the Series will be reflected in the DataFrame.

In [5]:
```
display(Image(filename="D:\\Python\\Python handbook\\6.jpg"))
```

Table 5-1. Possible data inputs to DataFrame constructor

Type	Notes
2D ndarray	A matrix of data, passing optional row and column labels
dict of arrays, lists, or tuples	Each sequence becomes a column in the DataFrame. All sequences must be the same length.
NumPy structured/record array	Treated as the "dict of arrays" case
dict of Series	Each value becomes a column. Indexes from each Series are unioned together to form the result's row index if no explicit index is passed.
dict of dicts	Each inner dict becomes a column. Keys are unioned to form the row index as in the "dict of Series" case.
list of dicts or Series	Each item becomes a row in the DataFrame. Union of dict keys or Series indexes become the DataFrame's column labels
List of lists or tuples	Treated as the "2D ndarray" case
Another DataFrame	The DataFrame's indexes are used unless different ones are passed
NumPy MaskedArray	Like the "2D ndarray" case except masked values become NA/missing in the DataFrame result

Index Objects

Pandas's Index objects are responsible for holding the axis labels and other metadata (like the axis

name or names). Any array or other sequence of labels used when constructing a `Series` *or* `DataFrame` *is internally converted to an Index:*

In [7]:
```
display(Image(filename="D:\\Python\\Python handbook\\7.jpg"))
```

Table 5-3. Index methods and properties

Method	Description
append	Concatenate with additional Index objects, producing a new Index
diff	Compute set difference as an Index
intersection	Compute set intersection
union	Compute set union
isin	Compute boolean array indicating whether each value is contained in the passed collection
delete	Compute new Index with element at index i deleted
drop	Compute new index by deleting passed values
insert	Compute new Index by inserting element at index i
is_monotonic	Returns True if each element is greater than or equal to the previous element
is_unique	Returns True if the Index has no duplicate values
unique	Compute the array of unique values in the Index

In [13]:
```
obj.index
```

Out[13]:
```
Index(['a', 'b', 'e', 'g'], dtype='object')
```

In [14]:
```
obj.unique
```

Out[14]:
```
<bound method Series.unique of a    4
b    7
e   -5
g    3
dtype: int64>
```

In [17]:
```
df.columns
```

Out[17]:
```
Index(['W', 'X', 'Y', 'Z', 'new'], dtype='object')
```

In [20]:
```
df.drop("new", axis = 1)
```

Out[20]:

	W	X	Y	Z
A	1.849800	1.387938	-1.137278	-0.123643
B	0.635781	-0.213673	0.444087	-1.345951
C	-1.188844	0.822141	-0.043321	1.888114
D	-0.871444	1.767228	0.566999	0.491481
E	0.249339	-1.767751	0.577606	-0.269734

In [21]:
```
df
```

Out[21]:

	W	X	Y	Z	new
A	1.849800	1.387938	-1.137278	-0.123643	0.712522
B	0.635781	-0.213673	0.444087	-1.345951	1.079867
C	-1.188844	0.822141	-0.043321	1.888114	-1.232164
D	-0.871444	1.767228	0.566999	0.491481	-0.304445
E	0.249339	-1.767751	0.577606	-0.269734	0.826946

The new column is still in the data frame to remove completly we need to use inplace argrment and the axis argument refers to the column or row if axis is 0 (default) it is row if axis is 1 it is column

In [22]:
```
df.drop("new", axis = 1, inplace = True)
```

In [23]:
```
df
```

Out[23]:

	W	X	Y	Z
A	1.849800	1.387938	-1.137278	-0.123643
B	0.635781	-0.213673	0.444087	-1.345951
C	-1.188844	0.822141	-0.043321	1.888114
D	-0.871444	1.767228	0.566999	0.491481
E	0.249339	-1.767751	0.577606	-0.269734

In [25]:
```
"W" in df.columns
```

Out[25]: True

In [26]:
```
"A" in df.index
```

Out[26]: True

In [27]:
```
"new" in df.columns
```

Out[27]: False

Essential Functionality

Reindexing

series.reindex(index, method, level, fill_value)

Parameters

index : array-like, optional New labels / index to conform to, should be specified using keywords. Preferably an Index object to avoid duplicating data.

method : {None, 'backfill'/'bfill', 'pad'/'ffill', 'nearest'} Method to use for filling holes in reindexed DataFrame. Please note: this is only applicable to DataFrames/Series with a monotonically increasing/decreasing index.

```
* None (default): don't fill gaps
* pad / ffill: Propagate last valid observation forward to next
  valid.
* backfill / bfill: Use next valid observation to fill gap.
* nearest: Use nearest valid observations to fill gap.
```

copy : bool, default True Return a new object, even if the passed indexes are the same.

level : int or name Broadcast across a level, matching Index values on the passed MultiIndex level.

fill_value : scalar, default np.NaN Value to use for missing values. Defaults to NaN, but can be any "compatible" value.

limit : int, default None Maximum number of consecutive elements to forward or backward fill.

tolerance : optional Maximum distance between original and new labels for inexact matches. The values of the index at the matching locations most satisfy the equation abs(index[indexer] - target) <= tolerance.

```
Tolerance may be a scalar value, which applies the same tolerance
to all values, or list-like, which applies variable tolerance per
element. List-like includes list, tuple, array, Series, and must be
the same size as the index and its dtype must exactly match the
index's type.)
```

In [34]:
```
obj.reindex(["a", "b", "c", "e", "f", "g"], fill_value = 0)  #This does not change t
```

Out[34]:
```
a    4
b    7
c    0
e   -5
f    0
g    3
dtype: int64
```

Selection, Filtering

In [44]:
```
df["W"]
```

Out[44]:
```
A    1.849800
B    0.635781
C   -1.188844
D   -0.871444
E    0.249339
Name: W, dtype: float64
```

Selecting subset of rows and columns

`df.loc[]`

Allowed inputs are:

- A single label.
- A list or array of labels, e.g. ['a', 'b', 'c'].
- A slice object with labels, e.g. 'a':'f'.

  .. warning:: Note that contrary to usual python slices, **both** the

  `start and the stop are included`
- A boolean array of the same length as the axis being sliced, e.g. [True, False, True].
- An alignable boolean Series. The index of the key will be aligned before masking.
- An alignable Index. The Index of the returned selection will be the input.
- A `callable` function with one argument (the calling Series or DataFrame) and that returns valid output for indexing (one of the above)

In [47]:
```
df.loc["A","W"]
```

Out[47]:
```
1.8498001060059204
```

In [49]:
```
df["W"]["A"]
```

Out[49]:
```
1.8498001060059204
```

In [51]:
```
df.loc[["A", "B"],["W","X"]]
```

Out[51]:

	W	X
A	1.849800	1.387938
B	0.635781	-0.213673

In [55]:
```
df.iloc[2]   #Purely integer-location based indexing for selection by position.
```

Out[55]:
```
W   -1.188844
X    0.822141
Y   -0.043321
Z    1.888114
Name: C, dtype: float64
```

Conditional Selection *An important feature of pandas is conditional selection using bracket notation, very similar to numpy:*

In [56]:
```
df > 0
```

Out[56]:

	W	X	Y	Z
A	True	True	False	False
B	True	False	True	False
C	False	True	False	True
D	False	True	True	True
E	True	False	True	False

In [57]:
```
df[df>0]                    #Returns the data frame for which the given condition i
```

Out[57]:

	W	X	Y	Z
A	1.849800	1.387938	NaN	NaN
B	0.635781	NaN	0.444087	NaN
C	NaN	0.822141	NaN	1.888114
D	NaN	1.767228	0.566999	0.491481
E	0.249339	NaN	0.577606	NaN

In [59]:
```
df[df["W"]>0]
```

Out[59]:

	W	X	Y	Z
A	1.849800	1.387938	-1.137278	-0.123643
B	0.635781	-0.213673	0.444087	-1.345951
E	0.249339	-1.767751	0.577606	-0.269734

In [62]:
```
df[(df["W"]>0) & (df["X"]<0)]
```

Out[62]:

	W	X	Y	Z
B	0.635781	-0.213673	0.444087	-1.345951
E	0.249339	-1.767751	0.577606	-0.269734

In [63]:
```
df[df["X"]<0]["W"]
```

Out[63]:
```
B    0.635781
E    0.249339
Name: W, dtype: float64
```

More operation on index

In [65]:
```
df.reset_index()                # reset the index to default values like this
```

Out[65]:

	index	W	X	Y	Z
0	A	1.849800	1.387938	-1.137278	-0.123643
1	B	0.635781	-0.213673	0.444087	-1.345951
2	C	-1.188844	0.822141	-0.043321	1.888114
3	D	-0.871444	1.767228	0.566999	0.491481
4	E	0.249339	-1.767751	0.577606	-0.269734

In [75]:
```
df["States"] = "CA NY WY OR CO".split()
df
```

Out[75]:

	W	X	Y	Z	States
A	0.783226	-0.260299	-0.258958	0.042018	CA
B	-1.190230	0.875860	0.433010	-1.240584	NY
C	-0.479875	1.013206	-0.462740	-0.981134	WY
D	0.399906	1.382583	-0.682202	0.175025	OR
E	0.153638	-0.229203	-0.439973	0.681553	CO

In [76]:

```
df.set_index("States", inplace = True)
df
```

Out[76]:

	W	X	Y	Z
States				
CA	0.783226	-0.260299	-0.258958	0.042018
NY	-1.190230	0.875860	0.433010	-1.240584
WY	-0.479875	1.013206	-0.462740	-0.981134
OR	0.399906	1.382583	-0.682202	0.175025
CO	0.153638	-0.229203	-0.439973	0.681553

Applying Function

In [79]:

```
df = pd.DataFrame({'col1':[1,2,3,4],'col2':[444,555,666,444],'col3':['abc','def','gh
df.head()
```

Out[79]:

	col1	col2	col3
0	1	444	abc
1	2	555	def
2	3	666	ghi
3	4	444	xyz

In [85]:

```
df[["col1", "col2"]].apply(lambda x:x**2)
```

Out[85]:

	col1	col2
0	1	197136
1	4	308025
2	9	443556
3	16	197136

In [86]:

```
df["col1"].sum()
```

Out[86]:

```
10
```

In [88]:
```
df["col1"].mean()
```

Out[88]:
```
2.5
```

In [89]:
```
df["col1"].min()
```

Out[89]:
```
1
```

The applicable all statistical funtions are shown below

In [90]:
```
display(Image(filename="D:\\Python\\Python handbook\\8.jpg"))
```

Table 5-10. Descriptive and summary statistics

Method	Description
count	Number of non-NA values
describe	Compute set of summary statistics for Series or each DataFrame column
min, max	Compute minimum and maximum values
argmin, argmax	Compute index locations (integers) at which minimum or maximum value obtained, respectively
idxmin, idxmax	Compute index values at which minimum or maximum value obtained, respectively
quantile	Compute sample quantile ranging from 0 to 1
sum	Sum of values
mean	Mean of values
median	Arithmetic median (50% quantile) of values
mad	Mean absolute deviation from mean value
var	Sample variance of values
std	Sample standard deviation of values
skew	Sample skewness (3rd moment) of values
kurt	Sample kurtosis (4th moment) of values
cumsum	Cumulative sum of values
cummin, cummax	Cumulative minimum or maximum of values, respectively
cumprod	Cumulative product of values
diff	Compute 1st arithmetic difference (useful for time series)
pct_change	Compute percent changes

Info on Unique values, values count, info of data frame and description

In [92]:
```
df["col2"].unique()
```

Out[92]:
```
array([444, 555, 666], dtype=int64)
```

In [94]:
```
df["col2"].nunique()
```

Out[94]:
```
3
```

In [96]:
```
df["col2"].value_counts()
```

Out[96]:
```
444    2
666    1
555    1
Name: col2, dtype: int64
```

In [97]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4 entries, 0 to 3
Data columns (total 3 columns):
 #   Column  Non-Null Count  Dtype 
---  ------  --------------  ----- 
 0   col1    4 non-null      int64 
 1   col2    4 non-null      int64 
 2   col3    4 non-null      object
dtypes: int64(2), object(1)
memory usage: 224.0+ bytes
```

In [98]:
```
df.describe()
```

Out[98]:

	col1	col2
count	4.000000	4.000000
mean	2.500000	527.250000
std	1.290994	106.274409
min	1.000000	444.000000
25%	1.750000	444.000000
50%	2.500000	499.500000
75%	3.250000	582.750000
max	4.000000	666.000000

Sorting and Reordering

```
df.sort_values(
    by,
    axis=0,
    ascending=True,
    inplace=False,
    kind='quicksort',
    na_position='last',
    ignore_index=False,
    key: 'ValueKeyFunc' = None,
)
```

by : str or list of str Name or list of names to sort by.

```
- if `axis` is 0 or `'index'` then `by` may contain index
  levels and/or column labels.
- if `axis` is 1 or `'columns'` then `by` may contain column
  levels and/or index labels.
```

axis : {0 or 'index', 1 or 'columns'}, default 0 Axis to be sorted.

ascending : bool or list of bool, default True

```
 Sort ascending vs. descending. Specify list for multiple sort orders.
If this is a list of bools, must match the length of the by.
```

inplace : bool, default False If True, perform operation in-place.

kind : {'quicksort', 'mergesort', 'heapsort'}, default 'quicksort' Choice of sorting algorithm. See also ndarray.np.sort for more information. `mergesort` is the only stable algorithm. For DataFrames, this option is only applied when sorting on a single column or label.

na_position : {'first', 'last'}, default 'last' Puts NaNs at the beginning if `first`; `last` puts NaNs at the end.

ignore_index : bool, default False If True, the resulting axis will be labeled 0, 1, …, n - 1

In [99]:
```
df.sort_values(by = "col2")
```

Out[99]:

	col1	col2	col3
0	1	444	abc
3	4	444	xyz
1	2	555	def
2	3	666	ghi

In [101…
```
df.sort_values(by = "col2", ascending=False)
```

Out[101…

	col1	col2	col3
2	3	666	ghi
1	2	555	def
0	1	444	abc
3	4	444	xyz

Correlation and Covariance

In [102…
```
df.corr()
```

Out[102…

	col1	col2
col1	1.00000	0.13484
col2	0.13484	1.00000

In [103…
```
df.cov()
```

Out[103…

	col1	col2
col1	1.666667	18.50
col2	18.500000	11294.25

Handling Missing Data

Missing data is common in most data analysis applications. One of the goals in designing pandas was to make working with missing data as painless as possible. Pandas uses the floating point value `NaN (Not a Number)` to represent missing data in both floating as well as in non-floating point arrays.

In [4]:
```
df1 = pd.DataFrame({'A':[1,2,np.nan],
                    'B':[5,np.nan,np.nan],
                    'C':[1,2,3]})

df1
```

Out[4]:

	A	B	C
0	1.0	5.0	1
1	2.0	NaN	2
2	NaN	NaN	3

Some pandas filling missing data methods

In [5]:
```
display(Image(filename="D:\\Python\\Python handbook\\9.jpg"))
```

Table 5-12. NA handling methods

Argument	Description
`dropna`	Filter axis labels based on whether values for each label have missing data, with varying thresholds for how much missing data to tolerate.
`fillna`	Fill in missing data with some value or using an interpolation method such as `'ffill'` or `'bfill'`.
`isnull`	Return like-type object containing boolean values indicating which values are missing / NA.
`notnull`	Negation of `isnull`.

In [6]:
```
df1.isnull()
```

Out[6]:

	A	B	C
0	False	False	False
1	False	True	False
2	True	True	False

```
df1.dropna(axis=0, how='any', thresh=None, subset=None, inplace=False)
```

Remove missing values.

Parameters

axis : {0 or 'index', 1 or 'columns'}, default 0 Determine if rows or columns which contain missing values are removed.

```
* 0, or 'index' : Drop rows which contain missing values.
* 1, or 'columns' : Drop columns which contain missing value.
```

how : {'any', 'all'}, default 'any' Determine if row or column is removed from DataFrame, when we have at least one NA or all NA.

```
    * 'any' : If any NA values are present, drop that row or column.
    * 'all' : If all values are NA, drop that row or column.
```

thresh : int, optional Require that many non-NA values.

subset : array-like, optional Labels along other axis to consider, e.g. if you are dropping rows these would be a list of columns to include.

inplace : bool, default False If True, do operation inplace and return None.

In [7]:
```
df1.dropna()
```

Out[7]:

	A	B	C
0	1.0	5.0	1

`df1.fillna(value=None, method=None, axis=None, inplace=False, limit=None, downcast=None,) -> 'Optional[DataFrame]'`

Fill NA/NaN values using the specified method.

Parameters

value : scalar, dict, Series, or DataFrame Value to use to fill holes (e.g. 0), alternately a dict/Series/DataFrame of values specifying which value to use for each index (for a Series) or column (for a DataFrame). Values not in the dict/Series/DataFrame will not be filled. This value cannot be a list.

method : {'backfill', 'bfill', 'pad', 'ffill', None}, default None Method to use for filling holes in reindexed Series pad / ffill: propagate last valid observation forward to next valid backfill / bfill: use next valid observation to fill gap.

axis : {0 or 'index', 1 or 'columns'} Axis along which to fill missing values.

inplace : bool, default False If True, fill in-place. Note: this will modify any other views on this object (e.g., a no-copy slice for a column in a DataFrame).

limit : int, default None If method is specified, this is the maximum number of consecutive NaN values to forward/backward fill. In other words, if there is a gap with more than this number of consecutive NaNs, it will only be partially filled. If method is not specified, this is the maximum number of entries along the entire axis where NaNs will be filled. Must be greater than 0 if not None.

downcast : dict, default is None A dict of item->dtype of what to downcast if possible, or the string 'infer' which will try to downcast to an appropriate equal type (e.g. float64 to int64 if possible).

In [9]:
```
df1.fillna(0)                    #Unchange untill inplace is true
```

Out[9]:

	A	B	C
0	1.0	5.0	1
1	2.0	0.0	2
2	0.0	0.0	3

Data Loading, Storage and File Formart

CSV Files

Loading

syntax: `pd.read_csv("Path to file")`

In [6]:
```
df2 = pd.read_csv("D:/Python/Python handbook/Ecommerce Purchases")
df2.head()
```

Out[6]:

	Address	Lot	AM or PM	Browser Info	Company	Credit Card	CC Exp Date	CC Security Code	Provi
0	16629 Pace Camp Apt. 448\nAlexisborough, NE 77...	46 in	PM	Opera/9.56. (X11; Linux x86_64; sl-SI) Presto/2...	Martinez-Herman	6011929061123406	02/20	900	JCI c
1	9374 Jasmine Spurs Suite 508\nSouth John, TN 8...	28 rn	PM	Opera/8.93. (Windows 98; Win 9x 4.90; en-US) Pr...	Fletcher, Richards and Whitaker	3337758169645356	11/18	561	Masterc
2	Unit 0065 Box 5052\nDPO AP 27450	94 vE	PM	Mozilla/5.0 (compatible; MSIE 9.0; Windows NT ...	Simpson, Williams and Pham	675957666125	08/19	699	JCI c
3	7780 Julia Fords\nNew Stacy, WA 45798	36 vm	PM	Mozilla/5.0 (Macintosh; Intel Mac OS X 10_8_0 ...	Williams, Marshall and Buchanan	6011578504430710	02/24	384	Disc
4	23012 Munoz Drive Suite 337\nNew Cynthia, TX 5...	20 IE	AM	Opera/9.58. (X11; Linux x86_64; it-IT) Presto/2...	Brown, Watson and Andrews	6011456623207998	10/25	678	Di Cl C Blar

Storing

syntax: `df2.to_csv("Path to file")`

Excel File

Loading

syntax: `pd.read_excel("Path to file")`

to read a particular sheet from file: `pd.read_excel("Path to file",sheet_name="sheet no")`

In [10]:
```
df3 = pd.read_excel("D:/Python/Phase 2/Projects and Exerises/Py-DS-ML-Bootcamp-maste
df3
```

Out[10]:

	a	b	c	d
0	0	1	2	3
1	4	5	6	7
2	8	9	10	11
3	12	13	14	15

Storing

`syntax: df3.to_excel("Path to file")`

to read a particular sheet from file: `df3.to_excel("Path to file",sheet_name="sheet no")`

HTML

You may need to install htmllib5,lxml, and BeautifulSoup4. In your terminal/command prompt run:

`conda install lxml`

`conda install html5lib`

`conda install BeautifulSoup4`

Then restart Jupyter Notebook. (or use pip install if you aren't using the Anaconda Distribution)

Pandas can read table tabs off of html. For example:

HTML Input

Pandas read_html function will read tables off of a webpage and return a list of DataFrame objects:

syntax: `pd.read_html("URL")`

Data Wrangling: concatenation, Mearging, Joining

In [6]:
```
df1 = pd.DataFrame({'A': ['A0', 'A1', 'A2', 'A3'],
                    'B': ['B0', 'B1', 'B2', 'B3'],
                    'C': ['C0', 'C1', 'C2', 'C3'],
                    'D': ['D0', 'D1', 'D2', 'D3']},
                    index=[0, 1, 2, 3])

df1
```

Out[6]:

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1

	A	B	C	D
2	A2	B2	C2	D2
3	A3	B3	C3	D3

In [7]:
```
df2 = pd.DataFrame({'A': ['A4', 'A5', 'A6', 'A7'],
                    'B': ['B4', 'B5', 'B6', 'B7'],
                    'C': ['C4', 'C5', 'C6', 'C7'],
                    'D': ['D4', 'D5', 'D6', 'D7']},
                     index=[4, 5, 6, 7])

df2
```

Out[7]:

	A	B	C	D
4	A4	B4	C4	D4
5	A5	B5	C5	D5
6	A6	B6	C6	D6
7	A7	B7	C7	D7

In [8]:
```
df3 = pd.DataFrame({'A': ['A8', 'A9', 'A10', 'A11'],
                    'B': ['B8', 'B9', 'B10', 'B11'],
                    'C': ['C8', 'C9', 'C10', 'C11'],
                    'D': ['D8', 'D9', 'D10', 'D11']},
                    index=[8, 9, 10, 11])

df3
```

Out[8]:

	A	B	C	D
8	A8	B8	C8	D8
9	A9	B9	C9	D9
10	A10	B10	C10	D10
11	A11	B11	C11	D11

Concatenation of two data frames

syntax: `pd.concat(objs, axis=0, join='outer', ignore_index: bool = False, keys=None, levels=None, names=None, verify_integrity: bool = False, sort: bool = False, copy: bool = True,)`

```
                    -> Union[ForwardRef('DataFrame'),
ForwardRef('Series')]
```

Docstring: Concatenate pandas objects along a particular axis with optional set logic along the other axes.

Can also add a layer of hierarchical indexing on the concatenation axis, which may be useful if the labels are the same (or overlapping) on the passed axis number.

Parameters

objs : a sequence or mapping of Series or DataFrame objects If a mapping is passed, the sorted keys will be used as the `keys` argument, unless it is passed, in which case the values will be selected (see below). Any None objects will be dropped silently unless they are all None in which case a ValueError will be raised.

axis : {0/'index', 1/'columns'}, default 0 The axis to concatenate along.

join : {'inner', 'outer'}, default 'outer' How to handle indexes on other axis (or axes).

ignore_index : bool, default False If True, do not use the index values along the concatenation axis. The resulting axis will be labeled 0, ..., n - 1. This is useful if you are concatenating objects where the concatenation axis does not have meaningful indexing information. Note the index values on the other axes are still respected in the join.

keys : sequence, default None If multiple levels passed, should contain tuples. Construct hierarchical index using the passed keys as the outermost level.

levels : list of sequences, default None Specific levels (unique values) to use for constructing a MultiIndex. Otherwise they will be inferred from the keys.

names : list, default None Names for the levels in the resulting hierarchical index.

verify_integrity : bool, default False Check whether the new concatenated axis contains duplicates. This can be very expensive relative to the actual data concatenation.

sort : bool, default False Sort non-concatenation axis if it is not already aligned when `join` is 'outer'. This has no effect when `join='inner'`, which already preserves the order of the non-concatenation axis.

copy : bool, default True If False, do not copy data unnecessarily.

In [9]:
```
pd.concat([df1,df2,df3])
```

Out[9]:

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3
4	A4	B4	C4	D4
5	A5	B5	C5	D5
6	A6	B6	C6	D6
7	A7	B7	C7	D7
8	A8	B8	C8	D8
9	A9	B9	C9	D9
10	A10	B10	C10	D10
11	A11	B11	C11	D11

In [10]:
```
pd.concat([df1,df2,df3], axis = 1)
```

Out[10]:

	A	B	C	D	A	B	C	D	A	B	C	D
0	A0	B0	C0	D0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
1	A1	B1	C1	D1	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2	A2	B2	C2	D2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
3	A3	B3	C3	D3	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
4	NaN	NaN	NaN	NaN	A4	B4	C4	D4	NaN	NaN	NaN	NaN
5	NaN	NaN	NaN	NaN	A5	B5	C5	D5	NaN	NaN	NaN	NaN
6	NaN	NaN	NaN	NaN	A6	B6	C6	D6	NaN	NaN	NaN	NaN
7	NaN	NaN	NaN	NaN	A7	B7	C7	D7	NaN	NaN	NaN	NaN
8	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	A8	B8	C8	D8
9	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	A9	B9	C9	D9
10	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	A10	B10	C10	D10
11	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	A11	B11	C11	D11

Mearging two data frames

syntax: `pd.merge(left, right, how: str = 'inner', on=None, left_on=None, right_on=None, left_index: bool = False, right_index: bool = False, sort: bool = False, suffixes=('_x', '_y'), copy: bool = True, indicator: bool = False, validate=None) -> 'DataFrame'`

Docstring: Merge DataFrame or named Series objects with a database-style join.

The join is done on columns or indexes. If joining columns on columns, the DataFrame indexes *will be ignored*. Otherwise if joining indexes on indexes or indexes on a column or columns, the index will be passed on. When performing a cross merge, no column specifications to merge on are allowed.

Parameters

left : DataFrame right : DataFrame or named Series Object to merge with.

how : {'left', 'right', 'outer', 'inner', 'cross'}, default 'inner' Type of merge to be performed.

```
* left: use only keys from left frame, similar to a SQL left outer
join;
  preserve key order.
* right: use only keys from right frame, similar to a SQL right outer
join;
  preserve key order.
* outer: use union of keys from both frames, similar to a SQL full
outer
  join; sort keys lexicographically.
* inner: use intersection of keys from both frames, similar to a SQL
inner
  join; preserve the order of the left keys.
* cross: creates the cartesian product from both frames, preserves the
```

```
order
  of the left keys.
```

on : label or list Column or index level names to join on. These must be found in both DataFrames. If `on` is None and not merging on indexes then this defaults to the intersection of the columns in both DataFrames.

left_on : label or list, or array-like Column or index level names to join on in the left DataFrame. Can also be an array or list of arrays of the length of the left DataFrame. These arrays are treated as if they are columns.

right_on : label or list, or array-like Column or index level names to join on in the right DataFrame. Can also be an array or list of arrays of the length of the right DataFrame. These arrays are treated as if they are columns.

left_index : bool, default False Use the index from the left DataFrame as the join key(s). If it is a MultiIndex, the number of keys in the other DataFrame (either the index or a number of columns) must match the number of levels.

right_index : bool, default False Use the index from the right DataFrame as the join key. Same caveats as left_index.

sort : bool, default False Sort the join keys lexicographically in the result DataFrame. If False, the order of the join keys depends on the join type (how keyword).

suffixes : list-like, default is ("_x", "_y") A length-2 sequence where each element is optionally a string indicating the suffix to add to overlapping column names in `left` and `right` respectively. Pass a value of `None` instead of a string to indicate that the column name from `left` or `right` should be left as-is, with no suffix. At least one of the values must not be None.

copy : bool, default True If False, avoid copy if possible.

indicator : bool or str, default False If True, adds a column to the output DataFrame called "_merge" with information on the source of each row. The column can be given a different name by providing a string argument. The column will have a Categorical type with the value of "left_only" for observations whose merge key only appears in the left DataFrame, "right_only" for observations whose merge key only appears in the right DataFrame, and "both" if the observation's merge key is found in both DataFrames.

validate : str, optional If specified, checks if merge is of specified type.

```
* "one_to_one" or "1:1": check if merge keys are unique in both left
and right datasets.
* "one_to_many" or "1:m": check if merge keys are unique in left
dataset.
* "many_to_one" or "m:1": check if merge keys are unique in right
dataset.
* "many_to_many" or "m:m": allowed, but does not result in checks.
```

In [12]:

```
left = pd.DataFrame({'key': ['K0', 'K1', 'K2', 'K3'],
                     'A': ['A0', 'A1', 'A2', 'A3'],
```

```
                    'B': ['B0', 'B1', 'B2', 'B3']})

right = pd.DataFrame({'key': ['K0', 'K1', 'K2', 'K3'],
                          'C': ['C0', 'C1', 'C2', 'C3'],
                          'D': ['D0', 'D1', 'D2', 'D3']})
```

In [13]:
```
pd.merge(left,right,how="inner",on="key")
```

Out[13]:

	key	A	B	C	D
0	K0	A0	B0	C0	D0
1	K1	A1	B1	C1	D1
2	K2	A2	B2	C2	D2
3	K3	A3	B3	C3	D3

In [14]:
```
left = pd.DataFrame({'key1': ['K0', 'K0', 'K1', 'K2'],
                     'key2': ['K0', 'K1', 'K0', 'K1'],
                        'A': ['A0', 'A1', 'A2', 'A3'],
                        'B': ['B0', 'B1', 'B2', 'B3']})

right = pd.DataFrame({'key1': ['K0', 'K1', 'K1', 'K2'],
                              'key2': ['K0', 'K0', 'K0', 'K0'],
                                 'C': ['C0', 'C1', 'C2', 'C3'],
                                 'D': ['D0', 'D1', 'D2', 'D3']})
```

In [15]:
```
left
```

Out[15]:

	key1	key2	A	B
0	K0	K0	A0	B0
1	K0	K1	A1	B1
2	K1	K0	A2	B2
3	K2	K1	A3	B3

In [16]:
```
right
```

Out[16]:

	key1	key2	C	D
0	K0	K0	C0	D0
1	K1	K0	C1	D1
2	K1	K0	C2	D2
3	K2	K0	C3	D3

In [17]:
```
pd.merge(left,right,how="left",on="key1")
```

Out[17]:

	key1	key2_x	A	B	key2_y	C	D
0	K0	K0	A0	B0	K0	C0	D0

	key1	key2_x	A	B	key2_y	C	D
1	K0	K1	A1	B1	K0	C0	D0
2	K1	K0	A2	B2	K0	C1	D1
3	K1	K0	A2	B2	K0	C2	D2
4	K2	K1	A3	B3	K0	C3	D3

In [18]:
```
pd.merge(left,right,how="right",on="key1")
```

Out[18]:

	key1	key2_x	A	B	key2_y	C	D
0	K0	K0	A0	B0	K0	C0	D0
1	K0	K1	A1	B1	K0	C0	D0
2	K1	K0	A2	B2	K0	C1	D1
3	K1	K0	A2	B2	K0	C2	D2
4	K2	K1	A3	B3	K0	C3	D3

In [20]:
```
pd.merge(left,right,how="outer",on=["key1","key2"])
```

Out[20]:

	key1	key2	A	B	C	D
0	K0	K0	A0	B0	C0	D0
1	K0	K1	A1	B1	NaN	NaN
2	K1	K0	A2	B2	C1	D1
3	K1	K0	A2	B2	C2	D2
4	K2	K1	A3	B3	NaN	NaN
5	K2	K0	NaN	NaN	C3	D3

Joining

`left.join(other, on=None, how='left', lsuffix='', rsuffix='', sort=False) -> 'DataFrame'`

Docstring: Join columns of another DataFrame.

Join columns with `other` DataFrame either on index or on a key column. Efficiently join multiple DataFrame objects by index at once by passing a list.

Parameters

other : DataFrame, Series, or list of DataFrame Index should be similar to one of the columns in this one. If a Series is passed, its name attribute must be set, and that will be used as the column name in the resulting joined DataFrame.

on : str, list of str, or array-like, optional Column or index level name(s) in the caller to join on the index in `other`, otherwise joins index-on-index. If multiple values given, the `other`

DataFrame must have a MultiIndex. Can pass an array as the join key if it is not already contained in the calling DataFrame. Like an Excel VLOOKUP operation.

how : {'left', 'right', 'outer', 'inner'}, default 'left' How to handle the operation of the two objects.

```
    * left: use calling frame's index (or column if on is specified)
    * right: use `other`'s index.
    * outer: form union of calling frame's index (or column if on is
    specified) with `other`'s index, and sort it.
      lexicographically.
    * inner: form intersection of calling frame's index (or column if on
    is specified) with `other`'s index, preserving the order of the
    calling's one.
```

lsuffix : str, default '' Suffix to use from left frame's overlapping columns.

rsuffix : str, default '' Suffix to use from right frame's overlapping columns.

sort : bool, default False Order result DataFrame lexicographically by the join key. If False, the order of the join key depends on the join type (how keyword).

In [21]:
```
left = pd.DataFrame({'A': ['A0', 'A1', 'A2'],
                     'B': ['B0', 'B1', 'B2']},
                      index=['K0', 'K1', 'K2'])

right = pd.DataFrame({'C': ['C0', 'C2', 'C3'],
                    'D': ['D0', 'D2', 'D3']},
                      index=['K0', 'K2', 'K3'])
```

In [23]:
```
left
```

Out[23]:

	A	B
K0	A0	B0
K1	A1	B1
K2	A2	B2

In [24]:
```
right
```

Out[24]:

	C	D
K0	C0	D0
K2	C2	D2
K3	C3	D3

In [25]:
```
left.join(right)
```

Out[25]:

	A	B	C	D
K0	A0	B0	C0	D0

	A	B	C	D
K1	A1	B1	NaN	NaN
K2	A2	B2	C2	D2

In [26]:
```
left.join(right, how="outer")
```

Out[26]:

	A	B	C	D
K0	A0	B0	C0	D0
K1	A1	B1	NaN	NaN
K2	A2	B2	C2	D2
K3	NaN	NaN	C3	D3

In [28]:
```
left.join(right, how="right")
```

Out[28]:

	A	B	C	D
K0	A0	B0	C0	D0
K2	A2	B2	C2	D2
K3	NaN	NaN	C3	D3

In [29]:
```
left.join(right, how="inner")
```

Out[29]:

	A	B	C	D
K0	A0	B0	C0	D0
K2	A2	B2	C2	D2

Groupby

The groupby method allows you to group rows of data together and call aggregate functions

`df.groupby(by=None, axis=0, level=None, as_index: 'bool' = True, sort: 'bool' = True, group_keys: 'bool' = True, squeeze: 'bool', observed: 'bool' = False, dropna: 'bool' = True) -> 'DataFrameGroupBy'`

Docstring: Group DataFrame using a mapper or by a Series of columns.

A groupby operation involves some combination of splitting the object, applying a function, and combining the results. This can be used to group large amounts of data and compute operations on these groups.

Parameters

by : mapping, function, label, or list of labels Used to determine the groups for the groupby. If `by` is a function, it's called on each value of the object's index. If a dict or Series is passed, the Series or dict VALUES will be used to determine the groups (the Series values are first aligned; see `.align()` method). If an ndarray is passed, the values are used as-is to determine the

groups. A label or list of labels may be passed to group by the columns in `self`. Notice that a tuple is interpreted as a (single) key.

axis : {0 or 'index', 1 or 'columns'}, default 0 Split along rows (0) or columns (1).

level : int, level name, or sequence of such, default None If the axis is a MultiIndex (hierarchical), group by a particular level or levels.

as_index : bool, default True For aggregated output, return object with group labels as the index. Only relevant for DataFrame input. as_index=False is effectively "SQL-style" grouped output.

sort : bool, default True Sort group keys. Get better performance by turning this off. Note this does not influence the order of observations within each group. Groupby preserves the order of rows within each group.

group_keys : bool, default True When calling apply, add group keys to index to identify pieces.

squeeze : bool, default False Reduce the dimensionality of the return type if possible, otherwise return a consistent type.

observed : bool, default False This only applies if any of the groupers are Categoricals. If True: only show observed values for categorical groupers. If False: show all values for categorical groupers.

dropna : bool, default True If True, and if group keys contain NA values, NA values together with row/column will be dropped. If False, NA values will also be treated as the key in groups

In [30]:
```
df =pd.DataFrame({'Company':['GOOG','GOOG','MSFT','MSFT','FB','FB'],
        'Person':['Sam','Charlie','Amy','Vanessa','Carl','Sarah'],
        'Sales':[200,120,340,124,243,350]})

df
```

Out[30]:

	Company	Person	Sales
0	GOOG	Sam	200
1	GOOG	Charlie	120
2	MSFT	Amy	340
3	MSFT	Vanessa	124
4	FB	Carl	243
5	FB	Sarah	350

In [38]:
```
df.groupby("Company").mean()
```

Out[38]:

Company	Sales
FB	296.5
GOOG	160.0
MSFT	232.0

In [39]:
```
df.groupby("Company").sum()
```

Out[39]:

Company	Sales
FB	593
GOOG	320
MSFT	464

Data Visualization

Mainly we are using three data visualization libraries they are

- Matplotlib
- Seaborn
- Plotly and cufflinks

Mainly the matplotlib and seaborn are concentrating on normal plots and plotly and cufflinks are for interactive plots

Matplotlib

Matplotlib is the "grandfather" library of data visualization with Python. It was created by John Hunter. He created it to try to replicate MatLab's (another programming language) plotting capabilities in Python. So if you happen to be familiar with matlab, matplotlib will feel natural to you.

It is an excellent 2D and 3D graphics library for generating scientific figures.

Some of the major Pros of Matplotlib are:

- Generally easy to get started for simple plots
- Support for custom labels and texts
- Great control of every element in a figure
- High-quality output in many formats
- Very customizable in general

Matplotlib allows you to create reproducible figures programmatically. Let's learn how to use it! Before continuing this lecture, I encourage you just to explore the official Matplotlib web page: http://matplotlib.org/

Installation

You'll need to install matplotlib first with either:

```
conda install matplotlib
```

or `pip install matplotlib`

Importing

In [4]:
```
```

```
import matplotlib.pyplot as plt
```

In [5]:
```
%matplotlib inline
```

This line used to show the plots in jupyter notebook for normal code use `plt.show()` at the end

Basic Example

Let's walk through a very simple example using two numpy arrays:

Example

Let's walk through a very simple example using two numpy arrays. You can also use lists, but most likely you'll be passing numpy arrays or pandas columns (which essentially also behave like arrays).

The data we want to plot:

In [3]:
```
import numpy as np
x = np.linspace(0, 5, 11)
y = x ** 2
```

In [4]:
```
x
```

Out[4]:
```
array([0. , 0.5, 1. , 1.5, 2. , 2.5, 3. , 3.5, 4. , 4.5, 5. ])
```

In [5]:
```
y
```

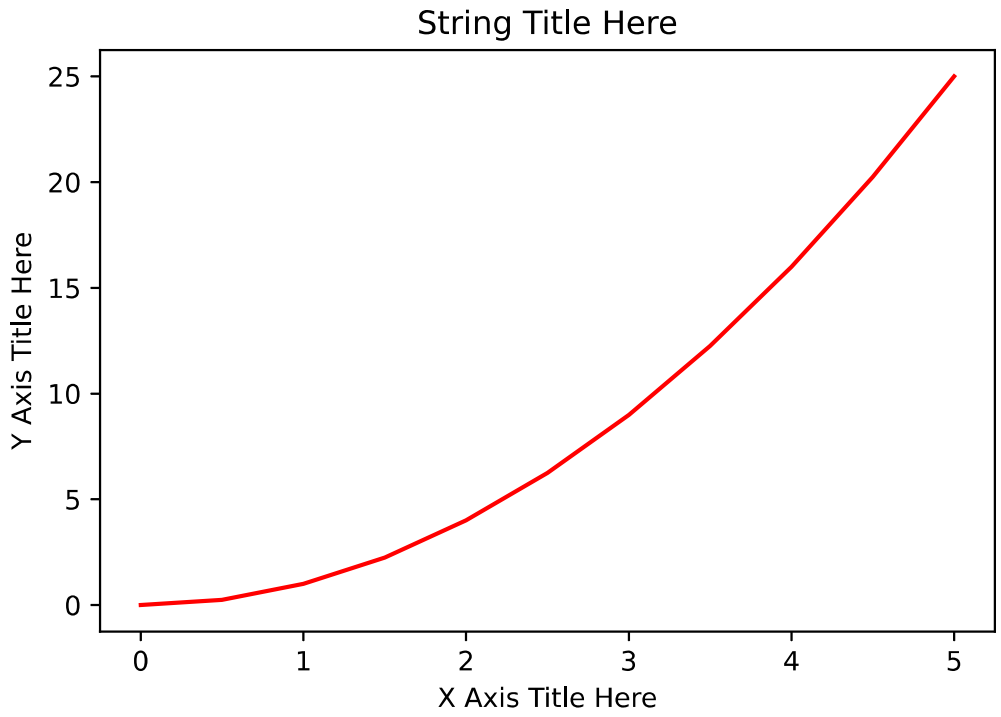
Out[5]:
```
array([ 0.  ,  0.25,  1.  ,  2.25,  4.  ,  6.25,  9.  , 12.25, 16.  ,
       20.25, 25.  ])
```

Basic Matplotlib Commands

We can create a very simple line plot using the following (I encourage you to pause and use Shift+Tab along the way to check out the document strings for the functions we are using).

In [ ]:
```
plt.plot()
plt.xlabel()
plt.title()
```

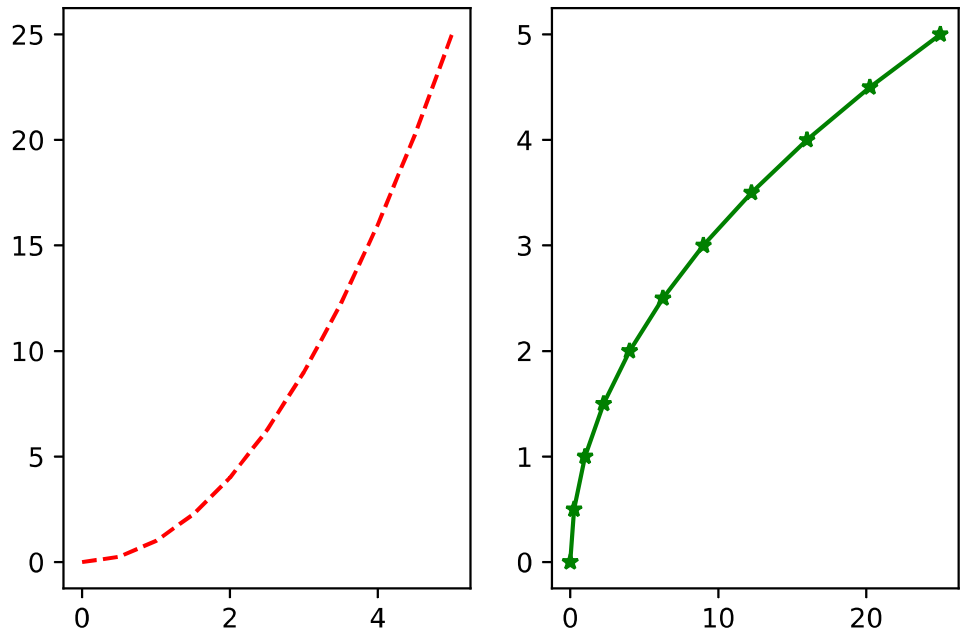
In [31]:
```
plt.plot(x, y, 'r') # 'r' is the color red
plt.xlabel('X Axis Title Here')
plt.ylabel('Y Axis Title Here')
plt.title('String Title Here')
plt.show()
```



Creating Multiplots on Same Canvas

In [ ]:
```
plt.subplot()
```

In [32]:
```
# plt.subplot(nrows, ncols, plot_number)
plt.subplot(1,2,1)
plt.plot(x, y, 'r--') # More on color options later
plt.subplot(1,2,2)
plt.plot(y, x, 'g*-');
```



Matplotlib Object Oriented Method

Now that we've seen the basics, let's break it all down with a more formal introduction of Matplotlib's Object Oriented API. This means we will instantiate figure objects and then call methods or attributes from that object.

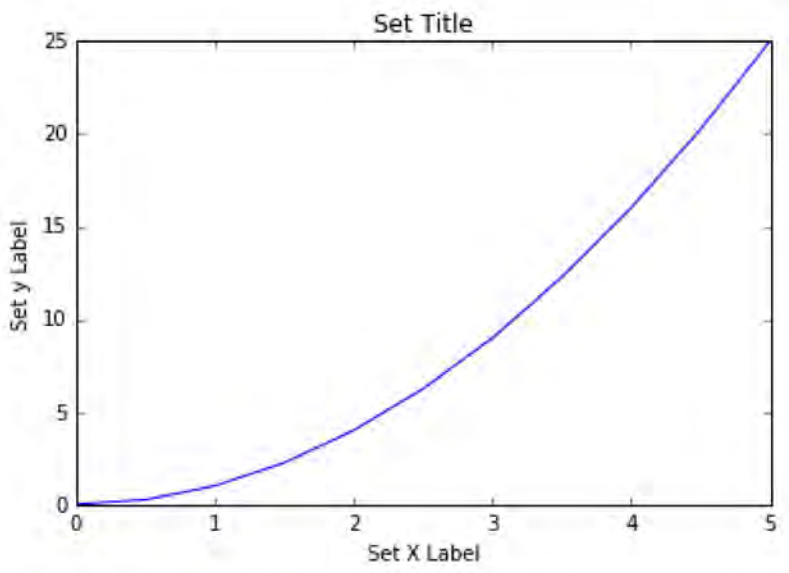
Introduction to the Object Oriented Method

The main idea in using the more formal Object Oriented method is to create figure objects and then just call methods or attributes off of that object. This approach is nicer when dealing with a canvas that has multiple plots on it.

To begin we create a figure instance. Then we can add axes to that figure:

In [15]:
```
# Create Figure (empty canvas)
fig = plt.figure()

# Add set of axes to figure
axes = fig.add_axes([0.1, 0.1, 0.8, 0.8]) # left, bottom, width, height (range 0 to

# Plot on that set of axes
axes.plot(x, y, 'b')
axes.set_xlabel('Set X Label') # Notice the use of set_ to begin methods
axes.set_ylabel('Set y Label')
axes.set_title('Set Title')
```

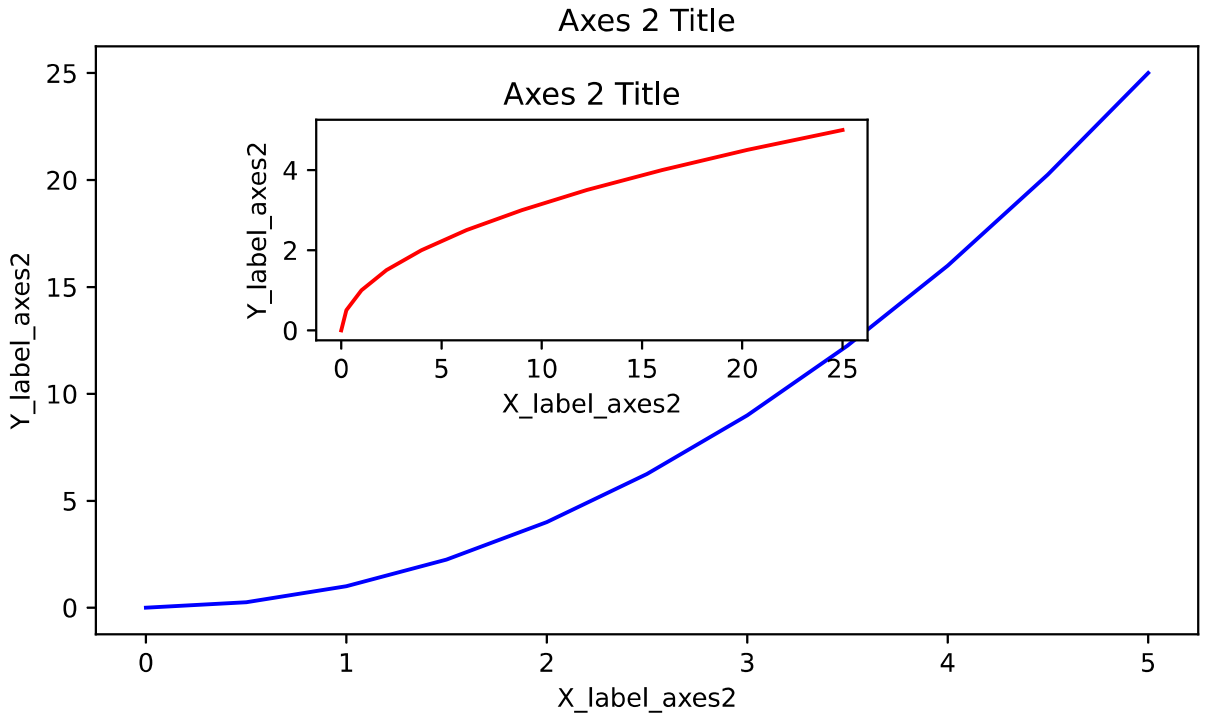
Out[15]:
```
<matplotlib.text.Text at 0x111c85198>
```



Code is a little more complicated, but the advantage is that we now have full control of where the plot axes are placed, and we can easily add more than one axis to the figure:

In [6]:
```
# Creates blank canvas
fig = plt.figure()

axes1 = fig.add_axes([0.1, 0.1, 1, 0.8]) # main axes
axes2 = fig.add_axes([0.3, 0.5, 0.5, 0.3]) # inset axes

# Larger Figure Axes 1
axes1.plot(x, y, 'b')
axes1.set_xlabel('X_label_axes2')
axes1.set_ylabel('Y_label_axes2')
axes1.set_title('Axes 2 Title')

# Insert Figure Axes 2
axes2.plot(y, x, 'r')
axes2.set_xlabel('X_label_axes2')
axes2.set_ylabel('Y_label_axes2')
axes2.set_title('Axes 2 Title')
```

Out[6]: Text(0.5, 1.0, 'Axes 2 Title')

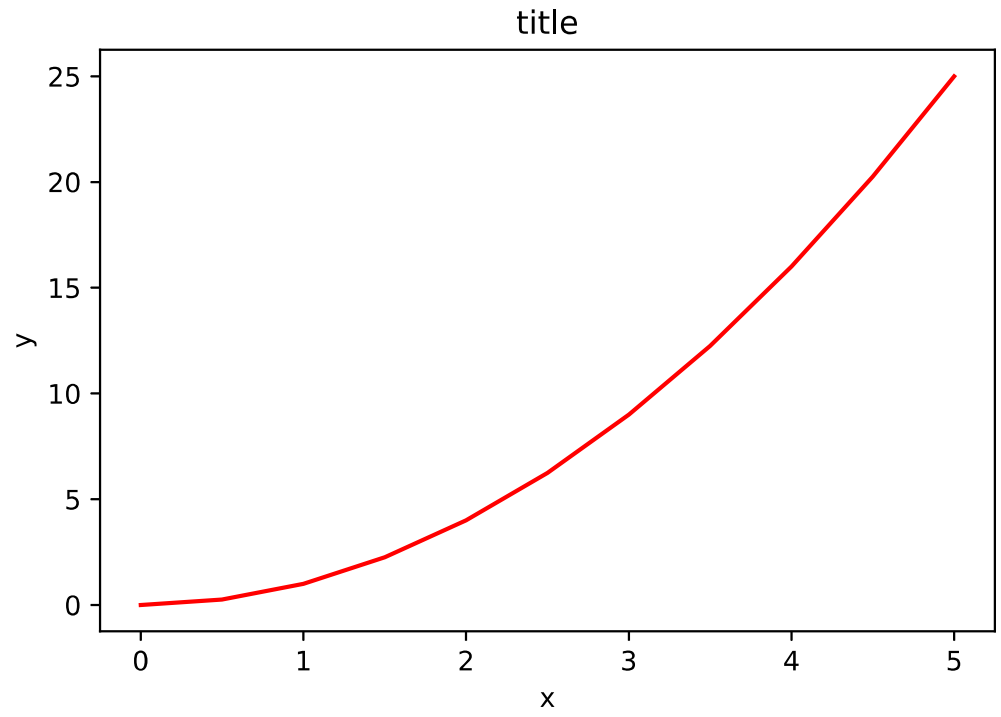


subplots()

The `plt.subplots()` object will act as a more automatic axis manager.

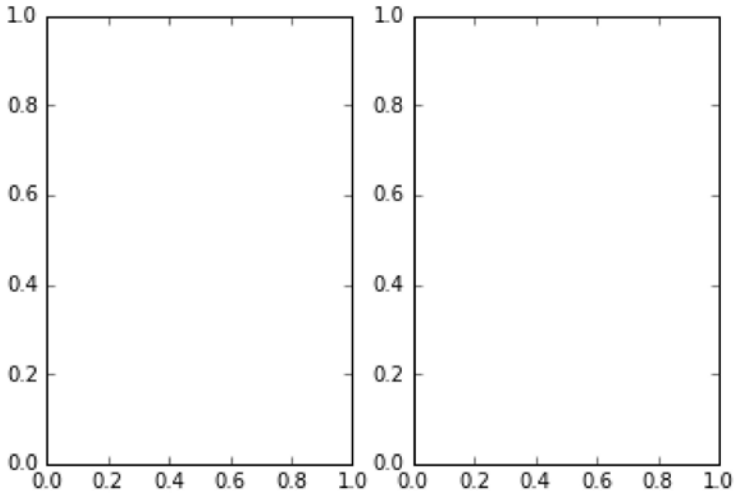
Basic use cases:

In [36]:
```
# Use similar to plt.figure() except use tuple unpacking to grab fig and axes
fig, axes = plt.subplots()

# Now use the axes object to add stuff to plot
axes.plot(x, y, 'r')
axes.set_xlabel('x')
axes.set_ylabel('y')
axes.set_title('title');
```



Then you can specify the number of rows and columns when creating the subplots() object:

In [24]:
```
# Empty canvas of 1 by 2 subplots
fig, axes = plt.subplots(nrows=1, ncols=2)
```



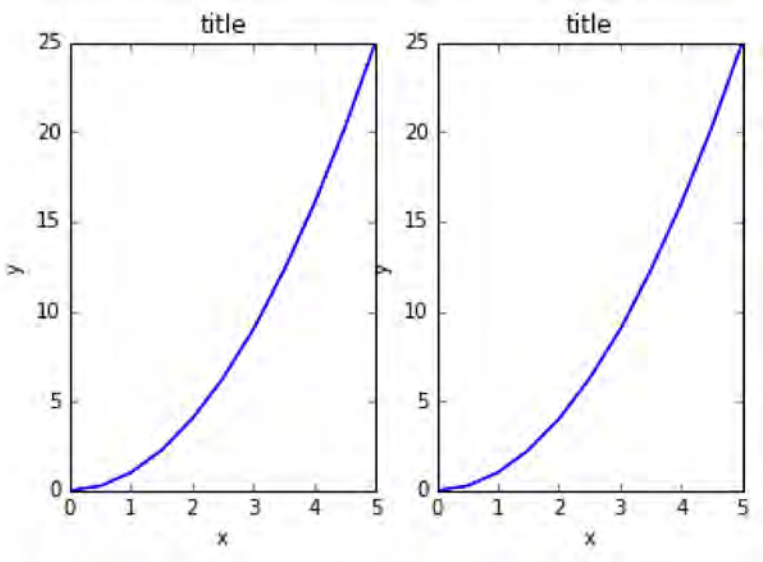
In [25]:
```
# Axes is an array of axes to plot on
axes
```

Out[25]:
```
array([<matplotlib.axes._subplots.AxesSubplot object at 0x111f0f8d0>,
       <matplotlib.axes._subplots.AxesSubplot object at 0x1121f5588>], dtype=object)
```

We can iterate through this array:

In [28]:
```
for ax in axes:
    ax.plot(x, y, 'b')
    ax.set_xlabel('x')
    ax.set_ylabel('y')
    ax.set_title('title')

# Display the figure object
fig
```

Out[28]:



A common issue with matplolib is overlapping subplots or figures. We ca use **fig.tight_layout()** or **plt.tight_layout()** method, which automatically adjusts the positions of the axes on the figure canvas so that there is no overlapping content:

In [32]:
```
fig, axes = plt.subplots(nrows=1, ncols=2)

for ax in axes:
    ax.plot(x, y, 'g')
    ax.set_xlabel('x')
    ax.set_ylabel('y')
    ax.set_title('title')

fig
plt.tight_layout()
```

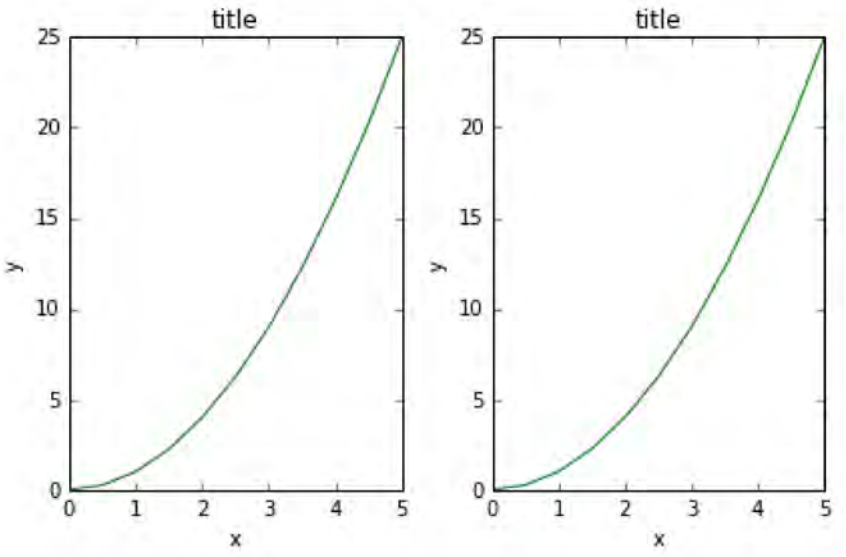


Figure size, aspect ratio and DPI

Matplotlib allows the aspect ratio, DPI and figure size to be specified when the Figure object is created. You can use the `figsize` and `dpi` keyword arguments.

- `figsize` is a tuple of the width and height of the figure in inches
- `dpi` is the dots-per-inch (pixel per inch).

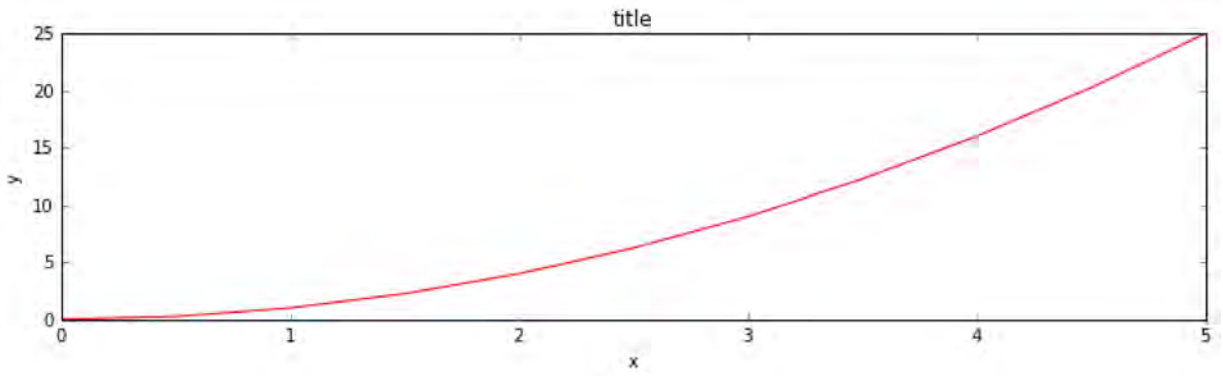
For example:

In [33]:
```
fig = plt.figure(figsize=(8,4), dpi=100)
```

```
<matplotlib.figure.Figure at 0x11228ea58>
```

The same arguments can also be passed to layout managers, such as the `subplots` function:

In [34]:
```
fig, axes = plt.subplots(figsize=(12,3))

axes.plot(x, y, 'r')
axes.set_xlabel('x')
axes.set_ylabel('y')
axes.set_title('title');
```



Saving figures

Matplotlib can generate high-quality output in a number formats, including PNG, JPG, EPS, SVG, PGF and PDF.

To save a figure to a file we can use the `savefig` method in the `Figure` class:

In [68]:
```
fig.savefig("filename.png")
```

Here we can also optionally specify the DPI and choose between different output formats:

In [69]:
```
fig.savefig("filename.png", dpi=200)
```

Legends, labels and titles

Now that we have covered the basics of how to create a figure canvas and add axes instances to the canvas, let's look at how decorate a figure with titles, axis labels, and legends.

Figure titles

A title can be added to each axis instance in a figure. To set the title, use the `set_title` method in the axes instance:

In [41]:
```
ax.set_title("title");
```

Axis labels

Similarly, with the methods `set_xlabel` and `set_ylabel`, we can set the labels of the X and Y axes:

In [42]:
```
ax.set_xlabel("x")
ax.set_ylabel("y");
```

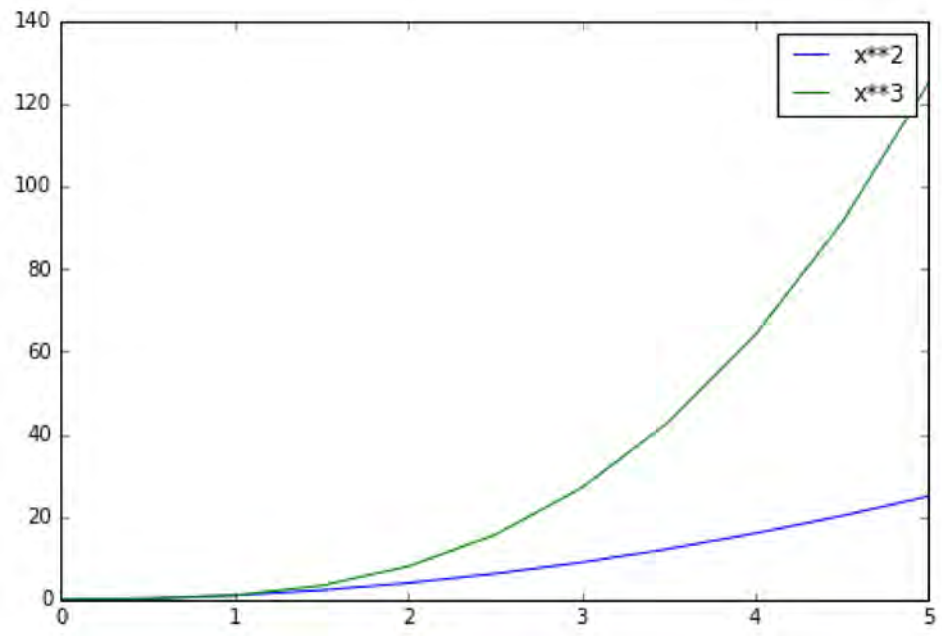
Legends

You can use the **label="label text"** keyword argument when plots or other objects are added to the figure, and then using the **legend** method without arguments to add the legend to the figure:

In [48]:
```
fig = plt.figure()

ax = fig.add_axes([0,0,1,1])

ax.plot(x, x**2, label="x**2")
```

```
ax.plot(x, x**3, label="x**3")
ax.legend()
```

Out[48]: <matplotlib.legend.Legend at 0x113a3d8d0>

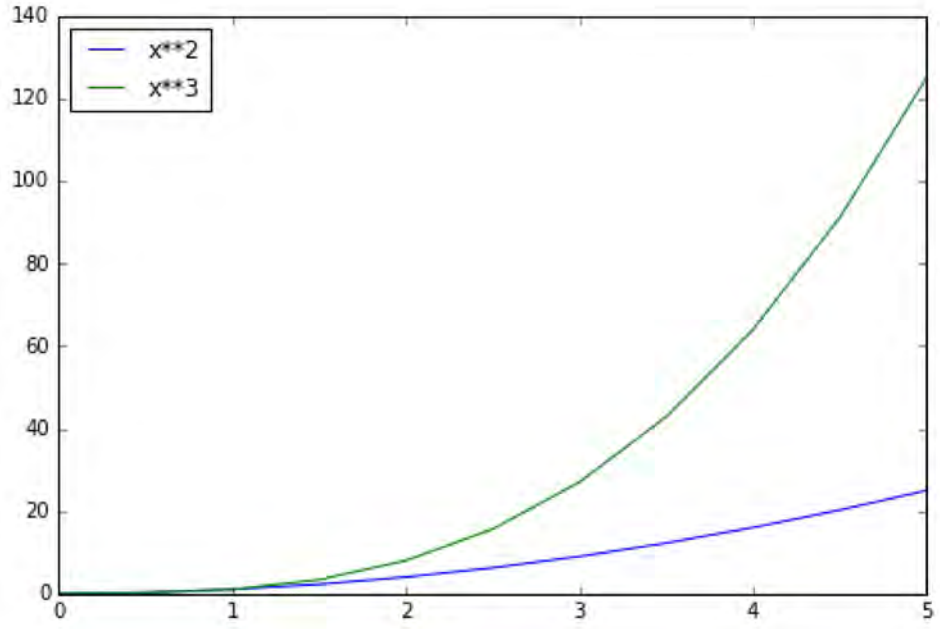


Notice how are legend overlaps some of the actual plot!

The **legend** function takes an optional keyword argument **loc** that can be used to specify where in the figure the legend is to be drawn. The allowed values of **loc** are numerical codes for the various places the legend can be drawn. See the documentation page for details. Some of the most common **loc** values are:

In [52]:
```
# Lots of options....

ax.legend(loc=1) # upper right corner
ax.legend(loc=2) # upper left corner
ax.legend(loc=3) # lower left corner
ax.legend(loc=4) # lower right corner

# .. many more options are available

# Most common to choose
ax.legend(loc=0) # let matplotlib decide the optimal location
fig
```

Out[52]:



Setting colors, linewidths, linetypes

Matplotlib gives you *a lot* of options for customizing colors, linewidths, and linetypes.

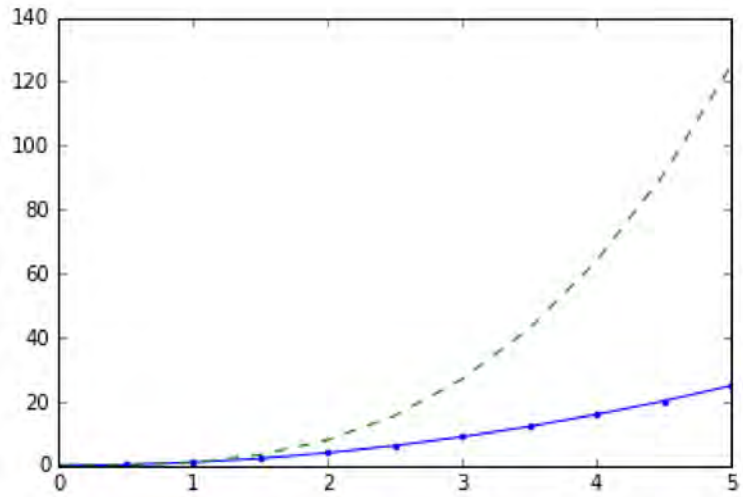
There is the basic MATLAB like syntax (which I would suggest you avoid using for more clairty sake:

Colors with MatLab like syntax

With matplotlib, we can define the colors of lines and other graphical elements in a number of ways. First of all, we can use the MATLAB-like syntax where `'b'` means blue, `'g'` means green, etc. The MATLAB API for selecting line styles are also supported: where, for example, 'b.-' means a blue line with dots:

In [54]:
```
# MATLAB style line color and style
fig, ax = plt.subplots()
ax.plot(x, x**2, 'b.-') # blue line with dots
ax.plot(x, x**3, 'g--') # green dashed line
```

Out[54]:
```
[<matplotlib.lines.Line2D at 0x111fae048>]
```

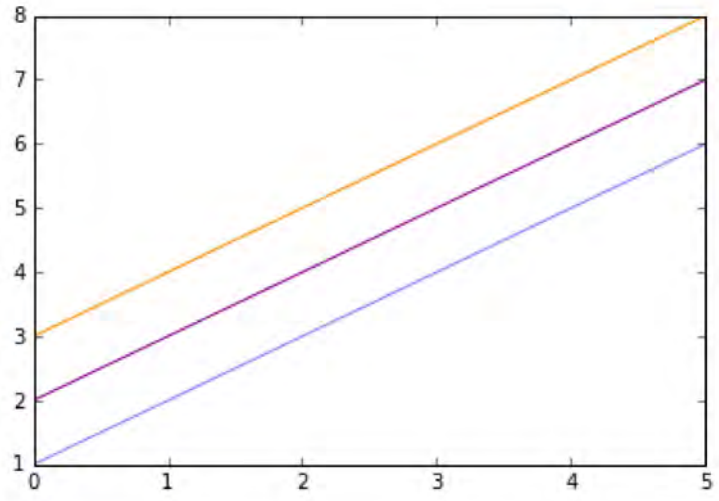


Colors with the color= parameter

We can also define colors by their names or RGB hex codes and optionally provide an alpha value using the `color` and `alpha` keyword arguments. Alpha indicates opacity.

In [56]:
```
fig, ax = plt.subplots()

ax.plot(x, x+1, color="blue", alpha=0.5) # half-transparant
ax.plot(x, x+2, color="#8B008B")        # RGB hex code
ax.plot(x, x+3, color="#FF8C00")        # RGB hex code
```

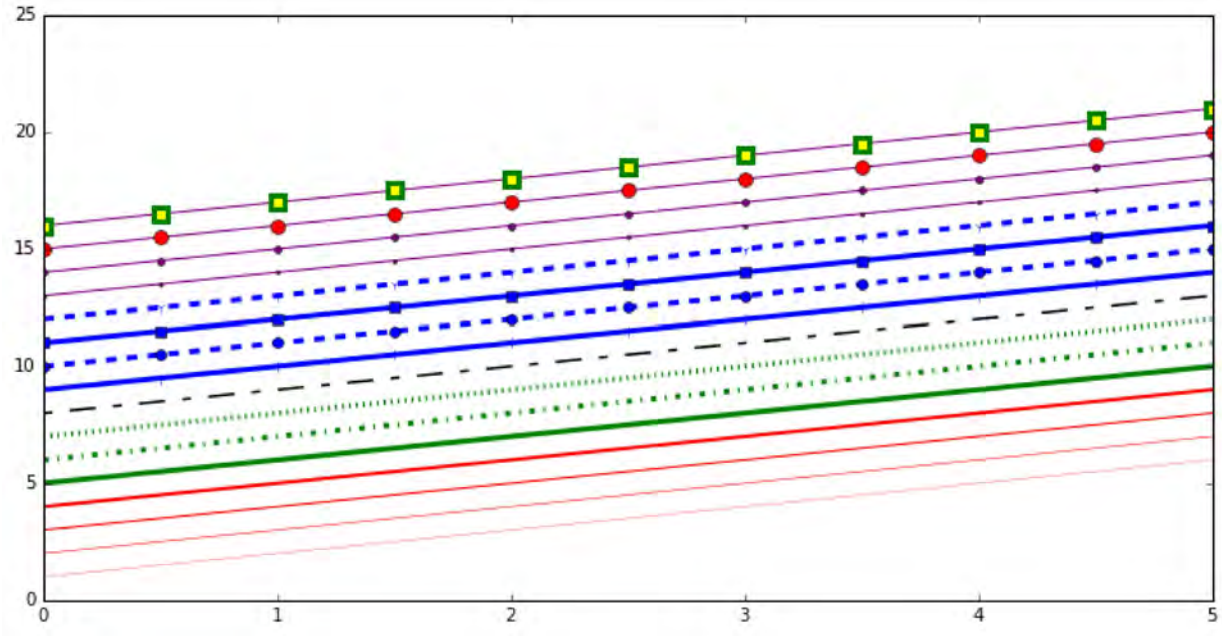
Out[56]:
```
[<matplotlib.lines.Line2D at 0x112179390>]
```



Line and marker styles

To change the line width, we can use the `linewidth` or `lw` keyword argument. The line style can be selected using the `linestyle` or `ls` keyword arguments:

In [57]:
```
fig, ax = plt.subplots(figsize=(12,6))

ax.plot(x, x+1, color="red", linewidth=0.25)
ax.plot(x, x+2, color="red", linewidth=0.50)
ax.plot(x, x+3, color="red", linewidth=1.00)
ax.plot(x, x+4, color="red", linewidth=2.00)

# possible linestype options ‘-‘, ‘–’, ‘-.’, ‘:’, ‘steps’
ax.plot(x, x+5, color="green", lw=3, linestyle='-')
ax.plot(x, x+6, color="green", lw=3, ls='-.')
ax.plot(x, x+7, color="green", lw=3, ls=':')

# custom dash
line, = ax.plot(x, x+8, color="black", lw=1.50)
line.set_dashes([5, 10, 15, 10]) # format: line length, space length, ...

# possible marker symbols: marker = '+', 'o', '*', 's', ',', '.', '1', '2', '3', '4'
ax.plot(x, x+ 9, color="blue", lw=3, ls='-', marker='+')
ax.plot(x, x+10, color="blue", lw=3, ls='--', marker='o')
ax.plot(x, x+11, color="blue", lw=3, ls='-', marker='s')
ax.plot(x, x+12, color="blue", lw=3, ls='--', marker='1')

# marker size and color
ax.plot(x, x+13, color="purple", lw=1, ls='-', marker='o', markersize=2)
ax.plot(x, x+14, color="purple", lw=1, ls='-', marker='o', markersize=4)
ax.plot(x, x+15, color="purple", lw=1, ls='-', marker='o', markersize=8, markerfacec
ax.plot(x, x+16, color="purple", lw=1, ls='-', marker='s', markersize=8,
        markerfacecolor="yellow", markeredgewidth=3, markeredgecolor="green");
```



Control over axis appearance

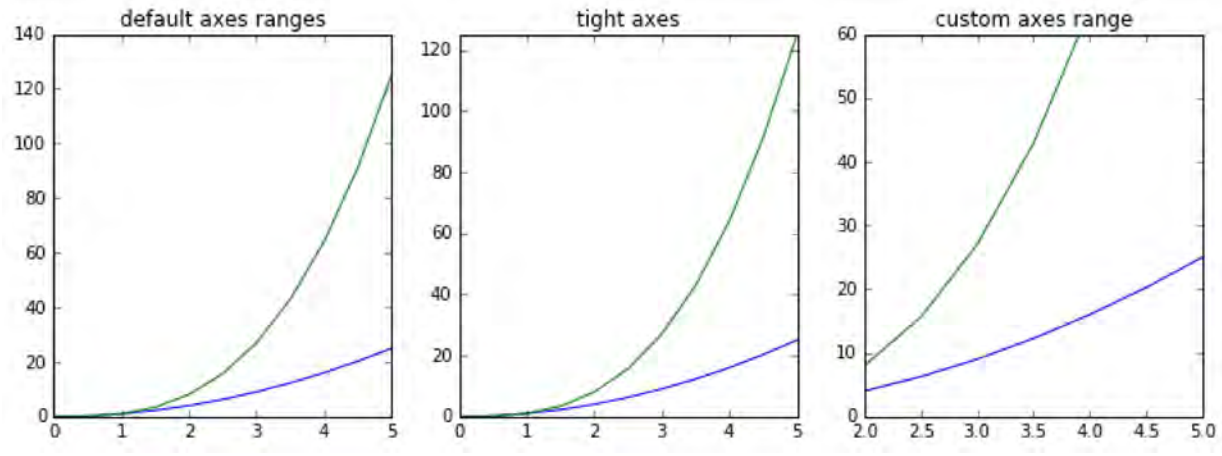
In this section we will look at controlling axis sizing properties in a matplotlib figure.

Plot range

We can configure the ranges of the axes using the `set_ylim` and `set_xlim` methods in the axis object, or `axis('tight')` for automatically getting "tightly fitted" axes ranges:

In [58]:

```
fig, axes = plt.subplots(1, 3, figsize=(12, 4))

axes[0].plot(x, x**2, x, x**3)
axes[0].set_title("default axes ranges")

axes[1].plot(x, x**2, x, x**3)
axes[1].axis('tight')
axes[1].set_title("tight axes")

axes[2].plot(x, x**2, x, x**3)
axes[2].set_ylim([0, 60])
axes[2].set_xlim([2, 5])
axes[2].set_title("custom axes range");
```



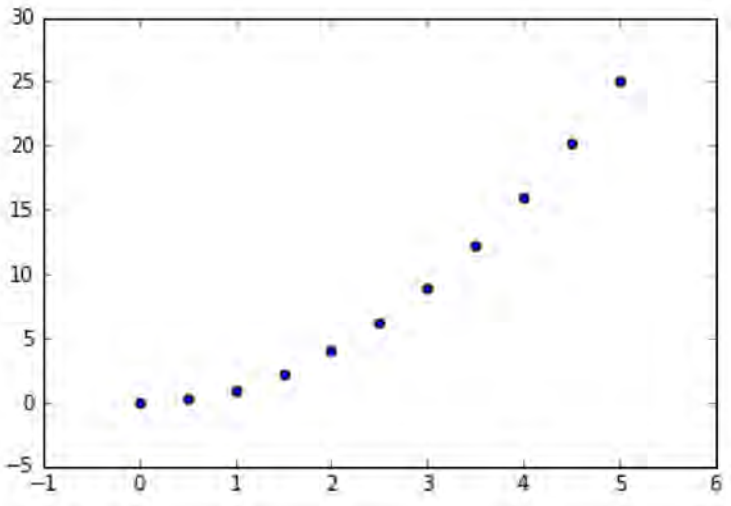
Special Plot Types

There are many specialized plots we can create, such as barplots, histograms, scatter plots, and much more. Most of these type of plots we will actually create using seaborn, a statistical

plotting library for Python. But here are a few examples of these type of plots:

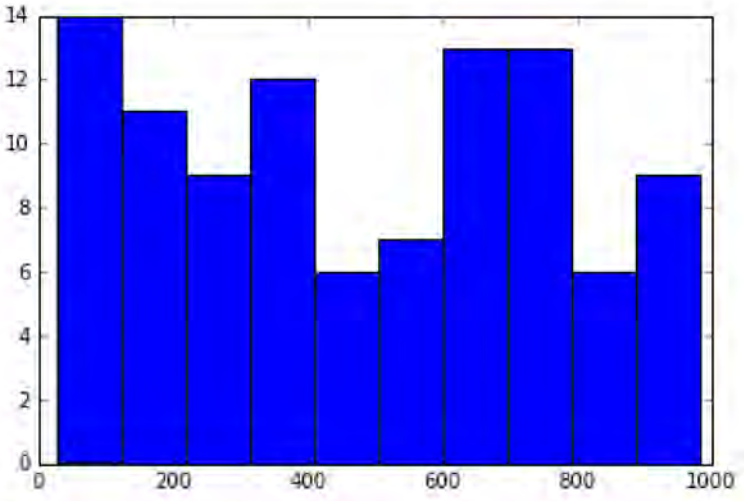
In [60]:
```
plt.scatter(x,y)
```

Out[60]:
```
<matplotlib.collections.PathCollection at 0x1122be438>
```

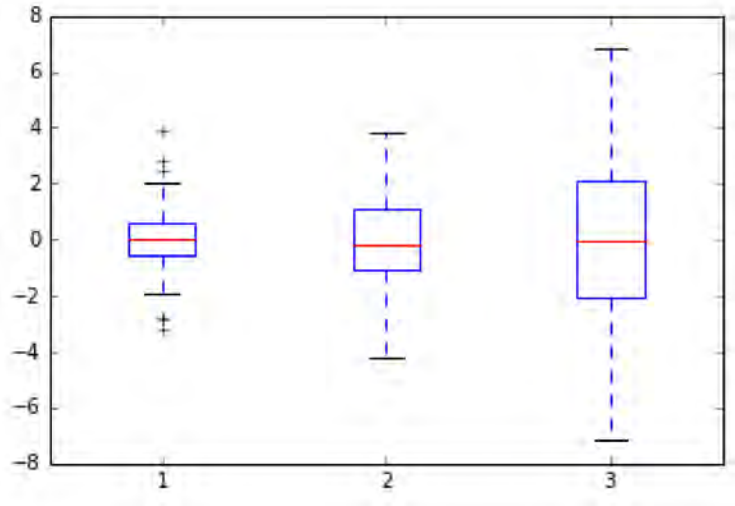


In [65]:
```
from random import sample
data = sample(range(1, 1000), 100)
plt.hist(data)
```

Out[65]:
```
(array([ 14.,  11.,   9.,  12.,   6.,   7.,  13.,  13.,   6.,   9.]),
 array([  28. ,  123.5,  219. ,  314.5,  410. ,  505.5,  601. ,  696.5,
         792. ,  887.5,  983. ]),
 <a list of 10 Patch objects>)
```



In [69]:
```
data = [np.random.normal(0, std, 100) for std in range(1, 4)]

# rectangular box plot
plt.boxplot(data,vert=True,patch_artist=True);
```



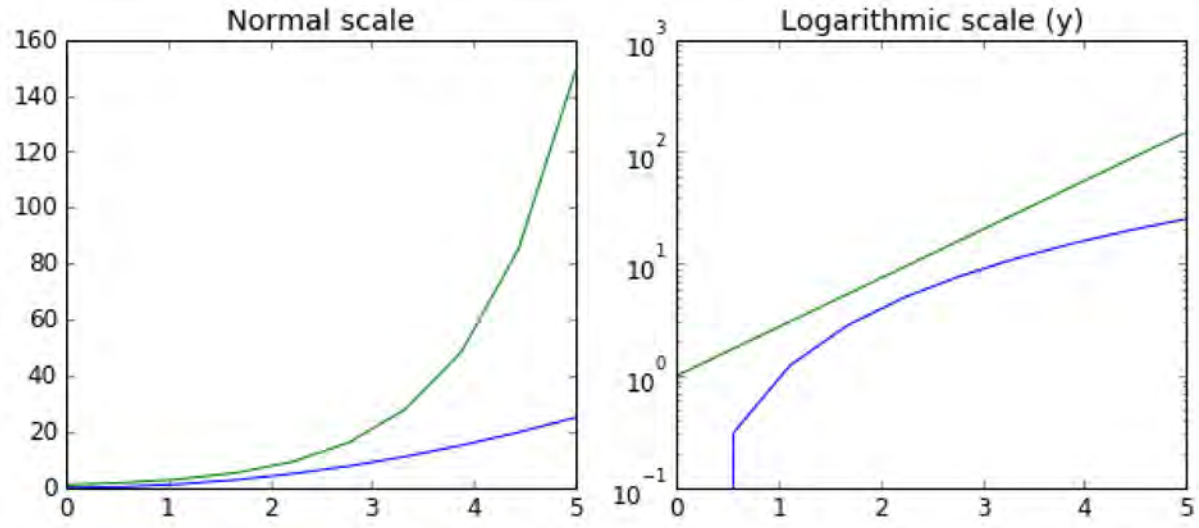
Further reading

- http://www.matplotlib.org - The project web page for matplotlib.
- https://github.com/matplotlib/matplotlib - The source code for matplotlib.
- http://matplotlib.org/gallery.html - A large gallery showcaseing various types of plots matplotlib can create. Highly recommended!
- http://www.loria.fr/~rougier/teaching/matplotlib - A good matplotlib tutorial.
- http://scipy-lectures.github.io/matplotlib/matplotlib.html - Another good matplotlib reference.

Logarithmic scale

It is also possible to set a logarithmic scale for one or both axes. This functionality is in fact only one application of a more general transformation system in Matplotlib. Each of the axes' scales are set seperately using `set_xscale` and `set_yscale` methods which accept one parameter (with the value "log" in this case):

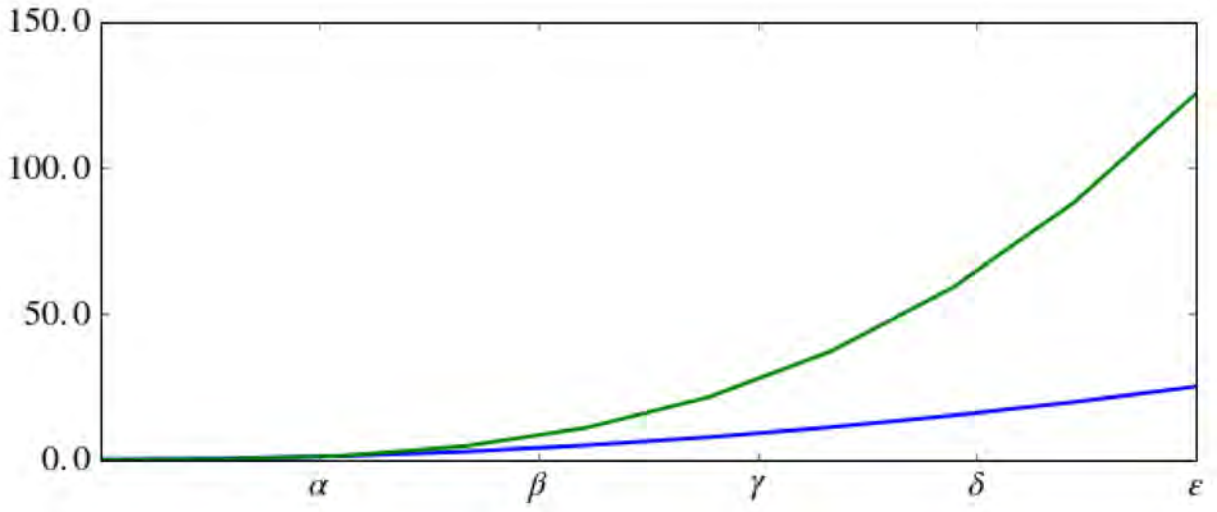
In [94]:
```
fig, axes = plt.subplots(1, 2, figsize=(10,4))

axes[0].plot(x, x**2, x, np.exp(x))
axes[0].set_title("Normal scale")

axes[1].plot(x, x**2, x, np.exp(x))
axes[1].set_yscale("log")
axes[1].set_title("Logarithmic scale (y)");
```



Placement of ticks and custom tick labels

We can explicitly determine where we want the axis ticks with `set_xticks` and `set_yticks`, which both take a list of values for where on the axis the ticks are to be placed. We can also use the `set_xticklabels` and `set_yticklabels` methods to provide a list of custom text labels for each tick location:

In [95]:
```
fig, ax = plt.subplots(figsize=(10, 4))

ax.plot(x, x**2, x, x**3, lw=2)

ax.set_xticks([1, 2, 3, 4, 5])
ax.set_xticklabels([r'$\alpha$', r'$\beta$', r'$\gamma$', r'$\delta$', r'$\epsilon$'

yticks = [0, 50, 100, 150]
ax.set_yticks(yticks)
ax.set_yticklabels(["$%.1f$" % y for y in yticks], fontsize=18); # use LaTeX formatt
```

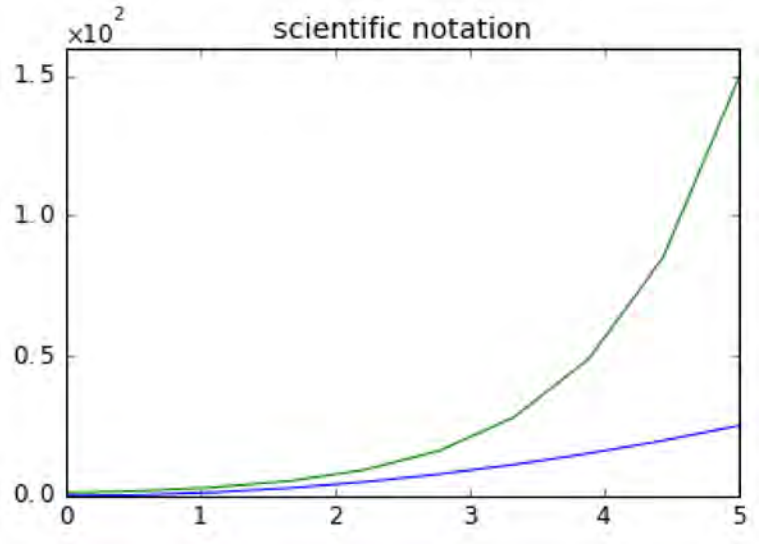


There are a number of more advanced methods for controlling major and minor tick placement in matplotlib figures, such as automatic placement according to different policies. See http://matplotlib.org/api/ticker_api.html for details.

Scientific notation

With large numbers on axes, it is often better use scientific notation:

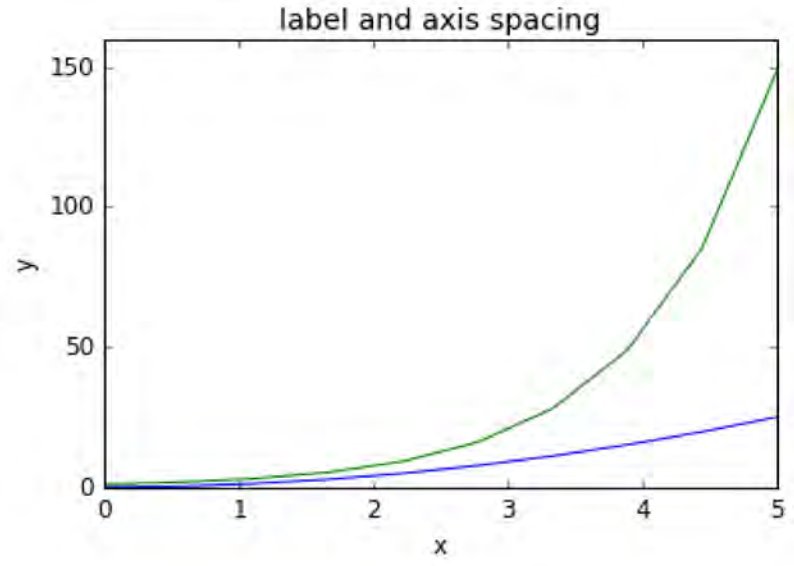
In [96]:
```
fig, ax = plt.subplots(1, 1)

ax.plot(x, x**2, x, np.exp(x))
ax.set_title("scientific notation")

ax.set_yticks([0, 50, 100, 150])

from matplotlib import ticker
formatter = ticker.ScalarFormatter(useMathText=True)
formatter.set_scientific(True)
formatter.set_powerlimits((-1,1))
ax.yaxis.set_major_formatter(formatter)
```



Axis number and axis label spacing

In [97]:

```
# distance between x and y axis and the numbers on the axes
matplotlib.rcParams['xtick.major.pad'] = 5
matplotlib.rcParams['ytick.major.pad'] = 5

fig, ax = plt.subplots(1, 1)

ax.plot(x, x**2, x, np.exp(x))
ax.set_yticks([0, 50, 100, 150])

ax.set_title("label and axis spacing")

# padding between axis label and axis numbers
ax.xaxis.labelpad = 5
ax.yaxis.labelpad = 5

ax.set_xlabel("x")
ax.set_ylabel("y");
```



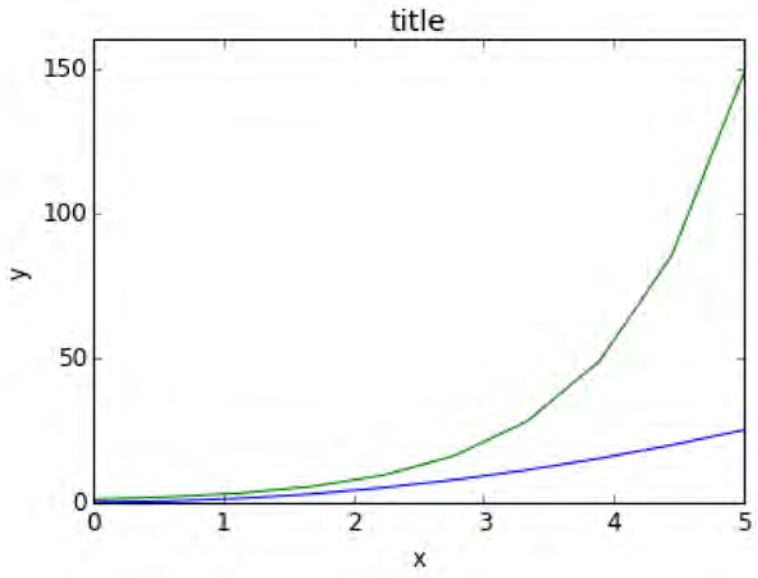
In [98]:

```
# restore defaults
matplotlib.rcParams['xtick.major.pad'] = 3
matplotlib.rcParams['ytick.major.pad'] = 3
```

Axis position adjustments

Unfortunately, when saving figures the labels are sometimes clipped, and it can be necessary to adjust the positions of axes a little bit. This can be done using `subplots_adjust`:

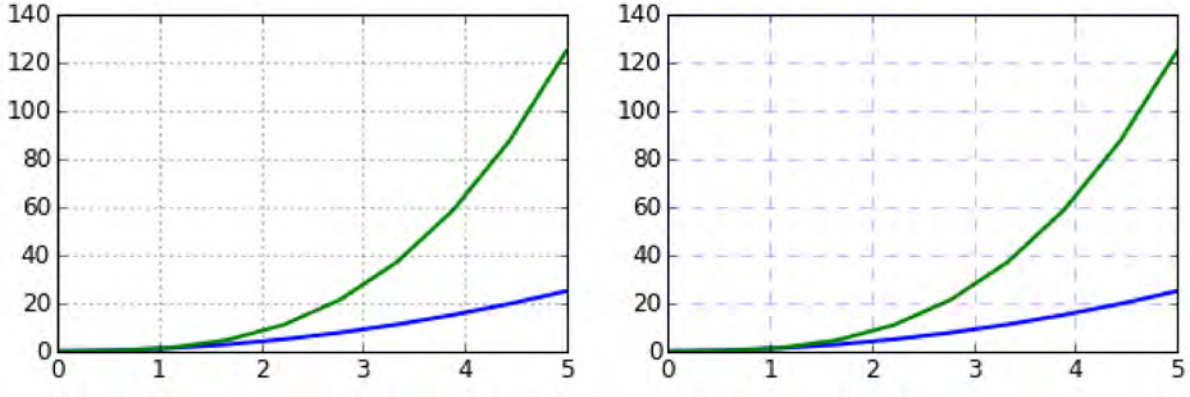
In [99]:
```
fig, ax = plt.subplots(1, 1)

ax.plot(x, x**2, x, np.exp(x))
ax.set_yticks([0, 50, 100, 150])

ax.set_title("title")
ax.set_xlabel("x")
ax.set_ylabel("y")

fig.subplots_adjust(left=0.15, right=.9, bottom=0.1, top=0.9);
```



Axis grid

With the `grid` method in the axis object, we can turn on and off grid lines. We can also customize the appearance of the grid lines using the same keyword arguments as the `plot` function:

In [100…
```
fig, axes = plt.subplots(1, 2, figsize=(10,3))

# default grid appearance
axes[0].plot(x, x**2, x, x**3, lw=2)
axes[0].grid(True)

# custom grid appearance
axes[1].plot(x, x**2, x, x**3, lw=2)
axes[1].grid(color='b', alpha=0.5, linestyle='dashed', linewidth=0.5)
```

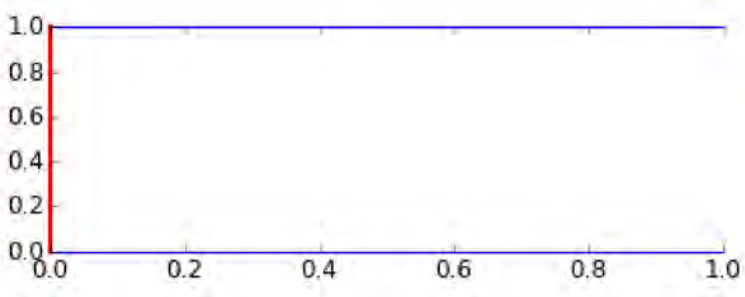


Axis spines

We can also change the properties of axis spines:

In [101…

```
fig, ax = plt.subplots(figsize=(6,2))

ax.spines['bottom'].set_color('blue')
ax.spines['top'].set_color('blue')

ax.spines['left'].set_color('red')
ax.spines['left'].set_linewidth(2)

# turn off axis spine to the right
ax.spines['right'].set_color("none")
ax.yaxis.tick_left() # only ticks on the left side
```

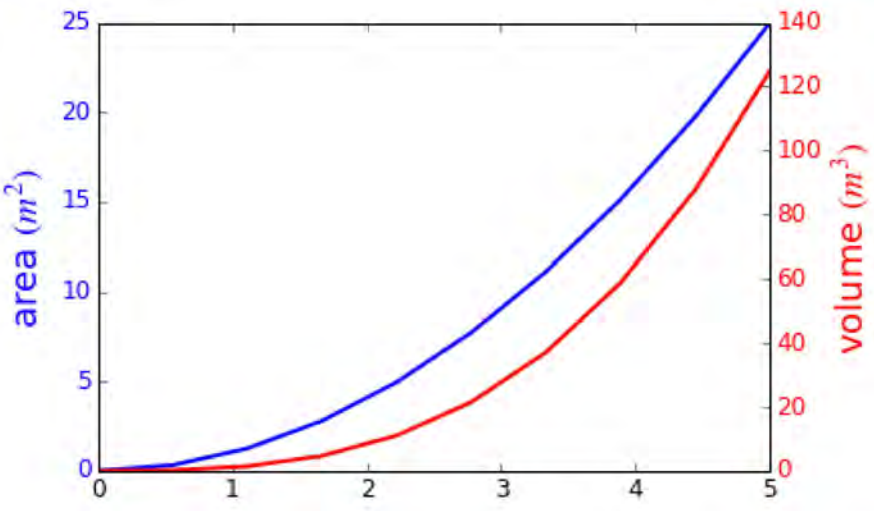


Twin axes

Sometimes it is useful to have dual x or y axes in a figure; for example, when plotting curves with different units together. Matplotlib supports this with the `twinx` and `twiny` functions:

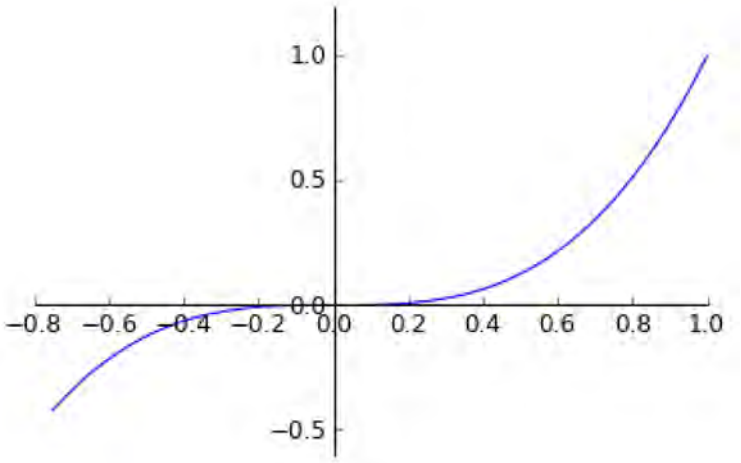
In [102…

```
fig, ax1 = plt.subplots()

ax1.plot(x, x**2, lw=2, color="blue")
ax1.set_ylabel(r"area $(m^2)$", fontsize=18, color="blue")
for label in ax1.get_yticklabels():
    label.set_color("blue")

ax2 = ax1.twinx()
ax2.plot(x, x**3, lw=2, color="red")
ax2.set_ylabel(r"volume $(m^3)$", fontsize=18, color="red")
for label in ax2.get_yticklabels():
    label.set_color("red")
```



Axes where x and y is zero

In [103…]
```
fig, ax = plt.subplots()

ax.spines['right'].set_color('none')
ax.spines['top'].set_color('none')

ax.xaxis.set_ticks_position('bottom')
ax.spines['bottom'].set_position(('data',0)) # set position of x spine to x=0

ax.yaxis.set_ticks_position('left')
ax.spines['left'].set_position(('data',0))   # set position of y spine to y=0

xx = np.linspace(-0.75, 1., 100)
ax.plot(xx, xx**3);
```

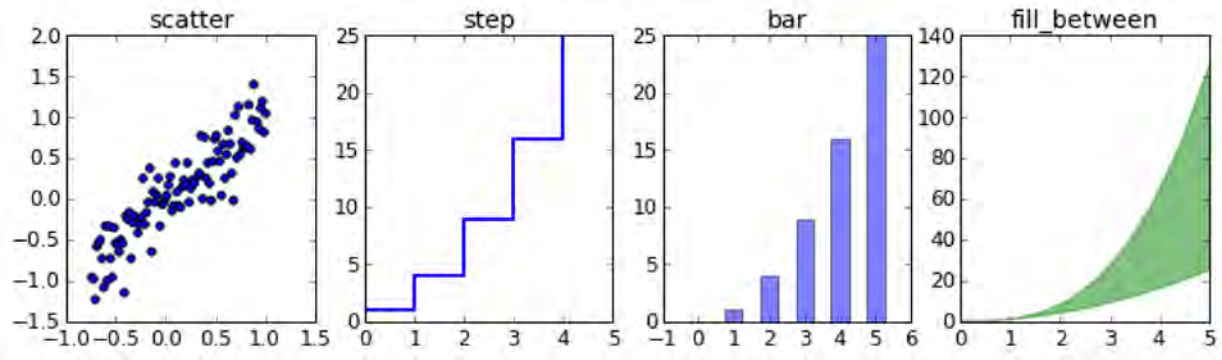


Other 2D plot styles

In addition to the regular `plot` method, there are a number of other functions for generating different kind of plots. See the matplotlib plot gallery for a complete list of available plot types: http://matplotlib.org/gallery.html. Some of the more useful ones are show below:

In [104…]
```
n = np.array([0,1,2,3,4,5])
```

In [105…]
```
fig, axes = plt.subplots(1, 4, figsize=(12,3))

axes[0].scatter(xx, xx + 0.25*np.random.randn(len(xx)))
axes[0].set_title("scatter")

axes[1].step(n, n**2, lw=2)
axes[1].set_title("step")

axes[2].bar(n, n**2, align="center", width=0.5, alpha=0.5)
axes[2].set_title("bar")

axes[3].fill_between(x, x**2, x**3, color="green", alpha=0.5);
axes[3].set_title("fill_between");
```

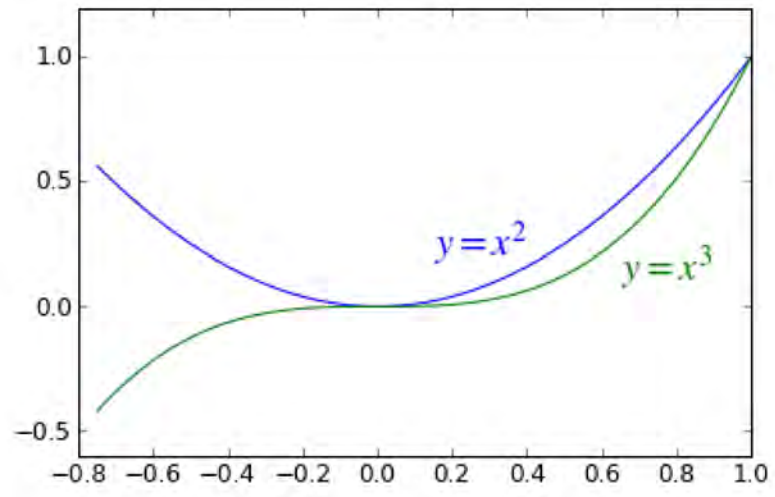


Text annotation

Annotating text in matplotlib figures can be done using the `text` function. It supports LaTeX formatting just like axis label texts and titles:

In [108…

```
fig, ax = plt.subplots()

ax.plot(xx, xx**2, xx, xx**3)

ax.text(0.15, 0.2, r"$y=x^2$", fontsize=20, color="blue")
ax.text(0.65, 0.1, r"$y=x^3$", fontsize=20, color="green");
```



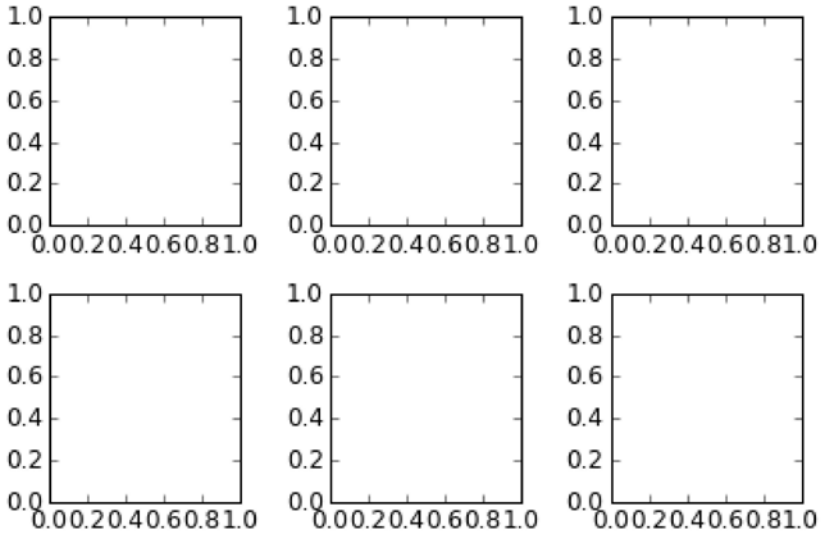
Figures with multiple subplots and insets

Axes can be added to a matplotlib Figure canvas manually using `fig.add_axes` or using a sub-figure layout manager such as `subplots`, `subplot2grid`, or `gridspec`:

subplots

In [109…

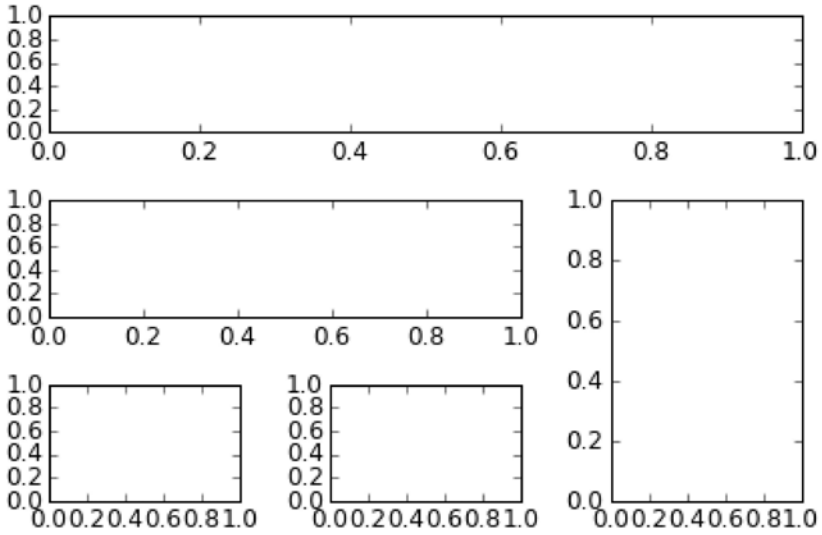
```
fig, ax = plt.subplots(2, 3)
fig.tight_layout()
```



subplot2grid

In [110…

```
fig = plt.figure()
ax1 = plt.subplot2grid((3,3), (0,0), colspan=3)
ax2 = plt.subplot2grid((3,3), (1,0), colspan=2)
ax3 = plt.subplot2grid((3,3), (1,2), rowspan=2)
ax4 = plt.subplot2grid((3,3), (2,0))
ax5 = plt.subplot2grid((3,3), (2,1))
fig.tight_layout()
```



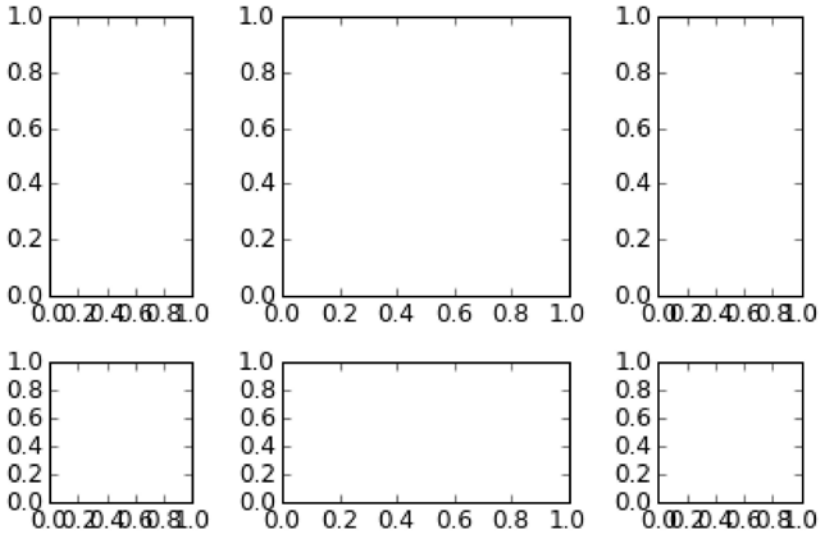
gridspec

In [111…

```
import matplotlib.gridspec as gridspec
```

In [112…

```
fig = plt.figure()

gs = gridspec.GridSpec(2, 3, height_ratios=[2,1], width_ratios=[1,2,1])
for g in gs:
    ax = fig.add_subplot(g)

fig.tight_layout()
```

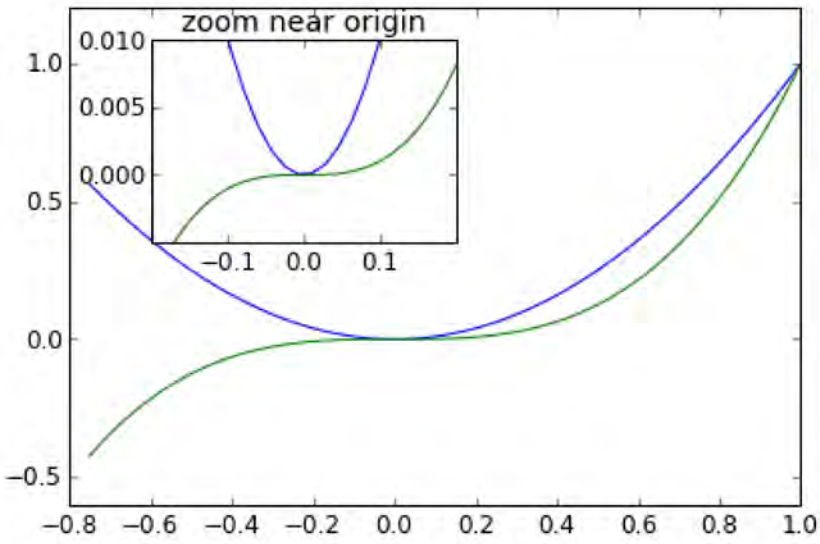


add_axes

Manually adding axes with `add_axes` is useful for adding insets to figures:

In [113…

```
fig, ax = plt.subplots()

ax.plot(xx, xx**2, xx, xx**3)
fig.tight_layout()

# inset
inset_ax = fig.add_axes([0.2, 0.55, 0.35, 0.35]) # X, Y, width, height

inset_ax.plot(xx, xx**2, xx, xx**3)
inset_ax.set_title('zoom near origin')

# set axis range
inset_ax.set_xlim(-.2, .2)
inset_ax.set_ylim(-.005, .01)

# set axis tick locations
inset_ax.set_yticks([0, 0.005, 0.01])
inset_ax.set_xticks([-0.1,0,.1]);
```



Colormap and contour figures

Colormaps and contour figures are useful for plotting functions of two variables. In most of these functions we will use a colormap to encode one dimension of the data. There are a

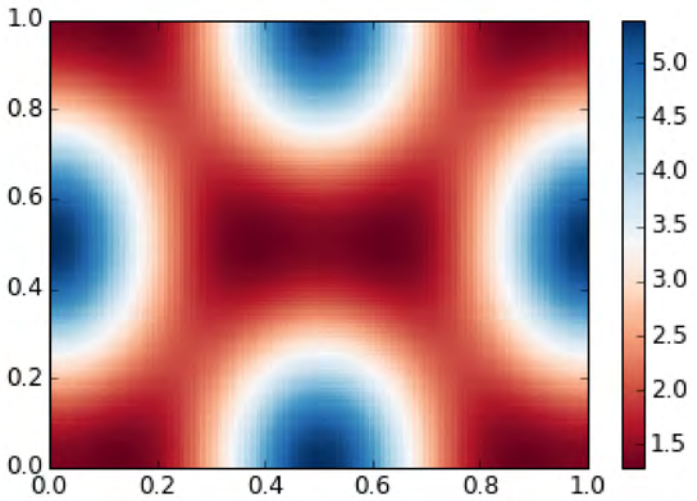
number of predefined colormaps. It is relatively straightforward to define custom colormaps. For a list of pre-defined colormaps, see:
http://www.scipy.org/Cookbook/Matplotlib/Show_colormaps

In [114…]
```
alpha = 0.7
phi_ext = 2 * np.pi * 0.5

def flux_qubit_potential(phi_m, phi_p):
    return 2 + alpha - 2 * np.cos(phi_p) * np.cos(phi_m) - alpha * np.cos(phi_ext -
```

In [115…]
```
phi_m = np.linspace(0, 2*np.pi, 100)
phi_p = np.linspace(0, 2*np.pi, 100)
X,Y = np.meshgrid(phi_p, phi_m)
Z = flux_qubit_potential(X, Y).T
```

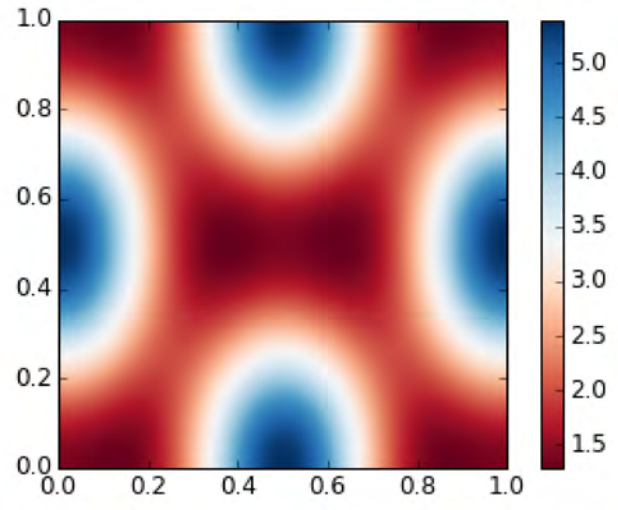
pcolor

In [116…]
```
fig, ax = plt.subplots()

p = ax.pcolor(X/(2*np.pi), Y/(2*np.pi), Z, cmap=matplotlib.cm.RdBu, vmin=abs(Z).min(
cb = fig.colorbar(p, ax=ax)
```



imshow

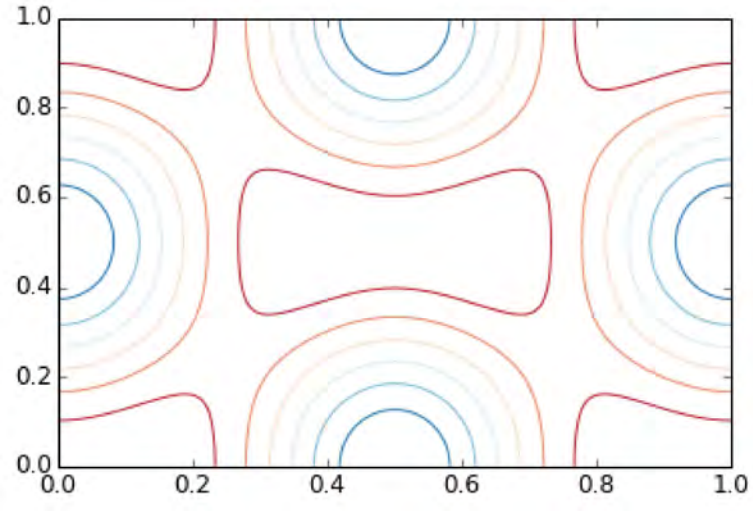
In [117…]
```
fig, ax = plt.subplots()

im = ax.imshow(Z, cmap=matplotlib.cm.RdBu, vmin=abs(Z).min(), vmax=abs(Z).max(), ext
im.set_interpolation('bilinear')

cb = fig.colorbar(im, ax=ax)
```



contour

In [118…

```
fig, ax = plt.subplots()

cnt = ax.contour(Z, cmap=matplotlib.cm.RdBu, vmin=abs(Z).min(), vmax=abs(Z).max(), e
```



3D figures

To use 3D graphics in matplotlib, we first need to create an instance of the `Axes3D` class. 3D axes can be added to a matplotlib figure canvas in exactly the same way as 2D axes; or, more conveniently, by passing a `projection='3d'` keyword argument to the `add_axes` or `add_subplot` methods.

In [119…

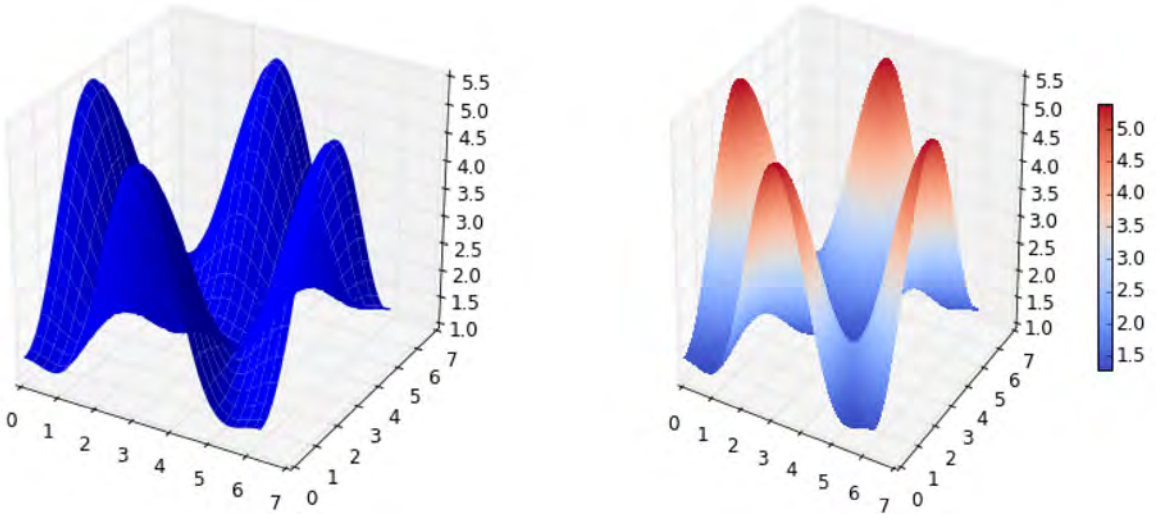
```
from mpl_toolkits.mplot3d.axes3d import Axes3D
```

Surface plots

In [121…

```
fig = plt.figure(figsize=(14,6))

# `ax` is a 3D-aware axis instance because of the projection='3d' keyword argument t
ax = fig.add_subplot(1, 2, 1, projection='3d')

p = ax.plot_surface(X, Y, Z, rstride=4, cstride=4, linewidth=0)

# surface_plot with color grading and color bar
ax = fig.add_subplot(1, 2, 2, projection='3d')
```

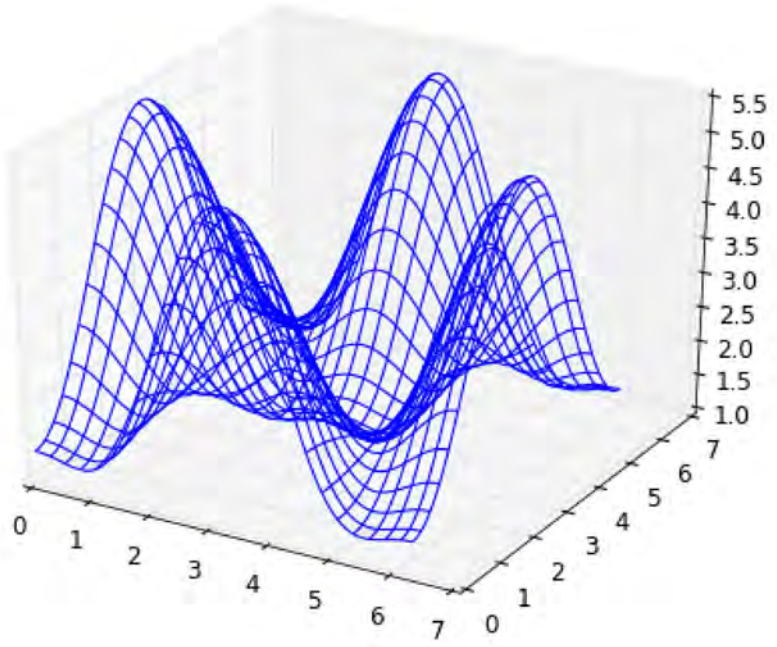
```
p = ax.plot_surface(X, Y, Z, rstride=1, cstride=1, cmap=matplotlib.cm.coolwarm, line
cb = fig.colorbar(p, shrink=0.5)
```



Wire-frame plot

In [122…

```
fig = plt.figure(figsize=(8,6))

ax = fig.add_subplot(1, 1, 1, projection='3d')

p = ax.plot_wireframe(X, Y, Z, rstride=4, cstride=4)
```

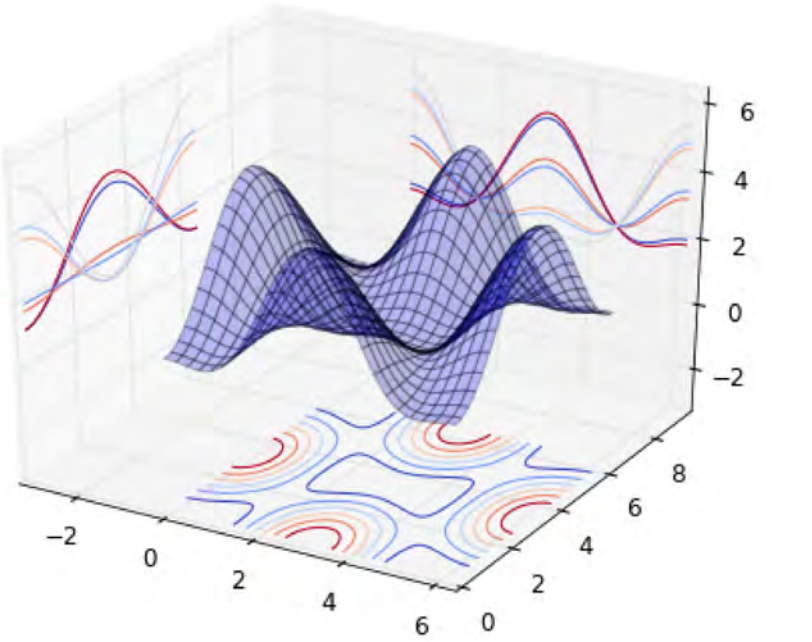


Coutour plots with projections

In [123…

```
fig = plt.figure(figsize=(8,6))

ax = fig.add_subplot(1,1,1, projection='3d')

ax.plot_surface(X, Y, Z, rstride=4, cstride=4, alpha=0.25)
cset = ax.contour(X, Y, Z, zdir='z', offset=-np.pi, cmap=matplotlib.cm.coolwarm)
cset = ax.contour(X, Y, Z, zdir='x', offset=-np.pi, cmap=matplotlib.cm.coolwarm)
cset = ax.contour(X, Y, Z, zdir='y', offset=3*np.pi, cmap=matplotlib.cm.coolwarm)
```

```
ax.set_xlim3d(-np.pi, 2*np.pi);
ax.set_ylim3d(0, 3*np.pi);
ax.set_zlim3d(-np.pi, 2*np.pi);
```



Further reading

- http://www.matplotlib.org - The project web page for matplotlib.
- https://github.com/matplotlib/matplotlib - The source code for matplotlib.
- http://matplotlib.org/gallery.html - A large gallery showcaseing various types of plots matplotlib can create. Highly recommended!
- http://www.loria.fr/~rougier/teaching/matplotlib - A good matplotlib tutorial.
- http://scipy-lectures.github.io/matplotlib/matplotlib.html - Another good matplotlib reference.

Distribution Plots

Let's discuss some plots that allow us to visualize the distribution of a data set. These plots are:

- distplot
- jointplot
- pairplot
- rugplot
- kdeplot

Imports

In [1]:
```
import seaborn as sns
%matplotlib inline
```

Data

Seaborn comes with built-in data sets!

In [3]:
```
tips = sns.load_dataset('tips')
```

In [5]:
```
tips
```

Out[5]:

	total_bill	tip	sex	smoker	day	time	size
0	16.99	1.01	Female	No	Sun	Dinner	2
1	10.34	1.66	Male	No	Sun	Dinner	3
2	21.01	3.50	Male	No	Sun	Dinner	3
3	23.68	3.31	Male	No	Sun	Dinner	2
4	24.59	3.61	Female	No	Sun	Dinner	4
...	...	...	...	...	...	...	...
239	29.03	5.92	Male	No	Sat	Dinner	3
240	27.18	2.00	Female	Yes	Sat	Dinner	2
241	22.67	2.00	Male	Yes	Sat	Dinner	2
242	17.82	1.75	Male	No	Sat	Dinner	2
243	18.78	3.00	Female	No	Thur	Dinner	2

244 rows × 7 columns

In [4]:
```
tips.head()
```

Out[4]:

	total_bill	tip	sex	smoker	day	time	size
0	16.99	1.01	Female	No	Sun	Dinner	2
1	10.34	1.66	Male	No	Sun	Dinner	3
2	21.01	3.50	Male	No	Sun	Dinner	3
3	23.68	3.31	Male	No	Sun	Dinner	2
4	24.59	3.61	Female	No	Sun	Dinner	4

distplot

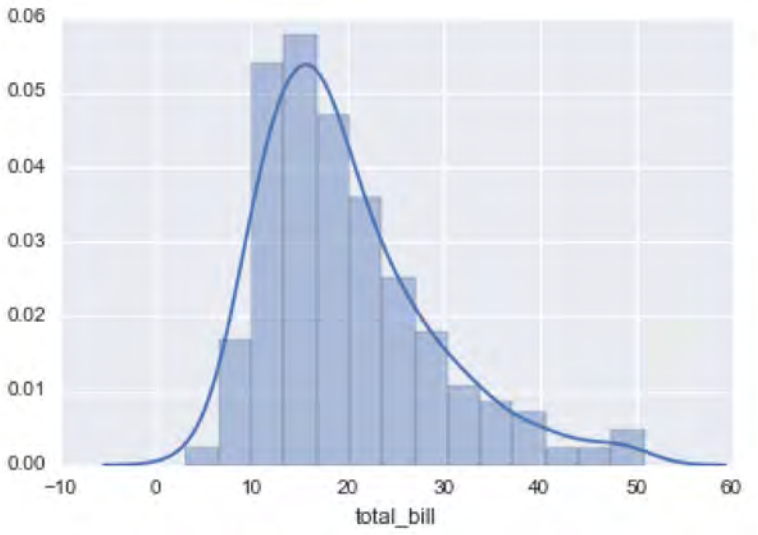
The distplot shows the distribution of a univariate set of observations.

In [ ]:
```
sns.distplot()
```

In [16]:
```
sns.distplot(tips['total_bill'])
# Safe to ignore warnings
```

```
/Users/marci/anaconda/lib/python3.5/site-packages/statsmodels/nonparametric/kdetool
s.py:20: VisibleDeprecationWarning: using a non-integer number instead of an integer
will result in an error in the future
  y = X[:m/2+1] + np.r_[0,X[m/2+1:],0]*1j
```

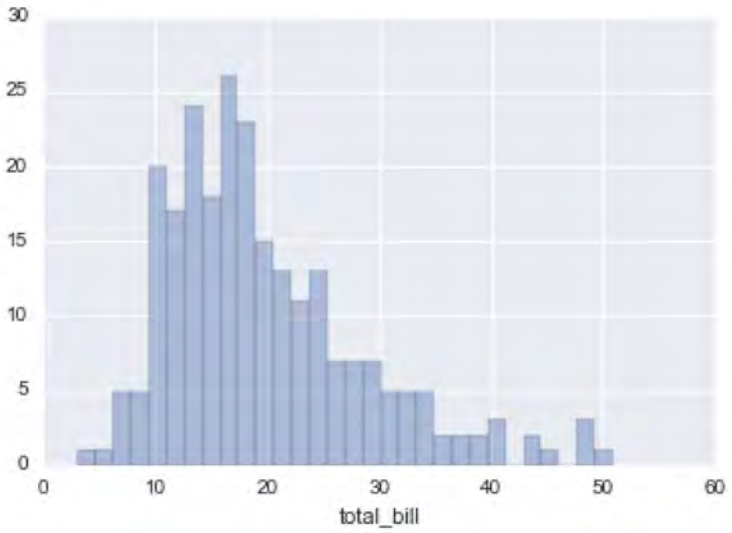
Out[16]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x11dd8e5f8>
```



To remove the kde layer and just have the histogram use:

In [9]:
```
sns.distplot(tips['total_bill'],kde=False,bins=30)
```

Out[9]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x11c7b8668>
```



jointplot

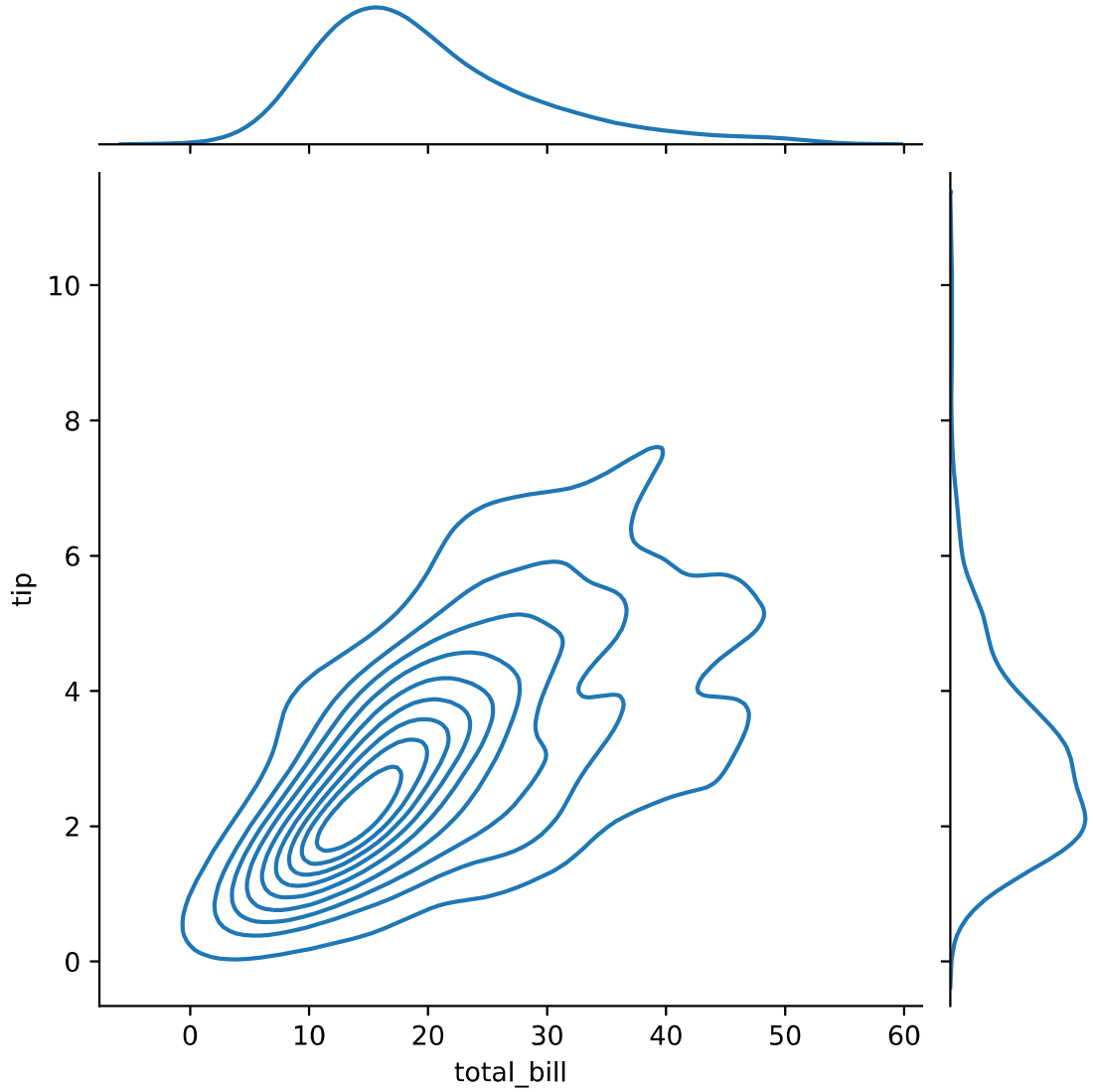
jointplot() allows you to basically match up two distplots for bivariate data. With your choice of what **kind** parameter to compare with:

- "scatter"
- "reg"
- "resid"
- "kde"
- "hex"

In [ ]:
```
sns.jointplot()
```

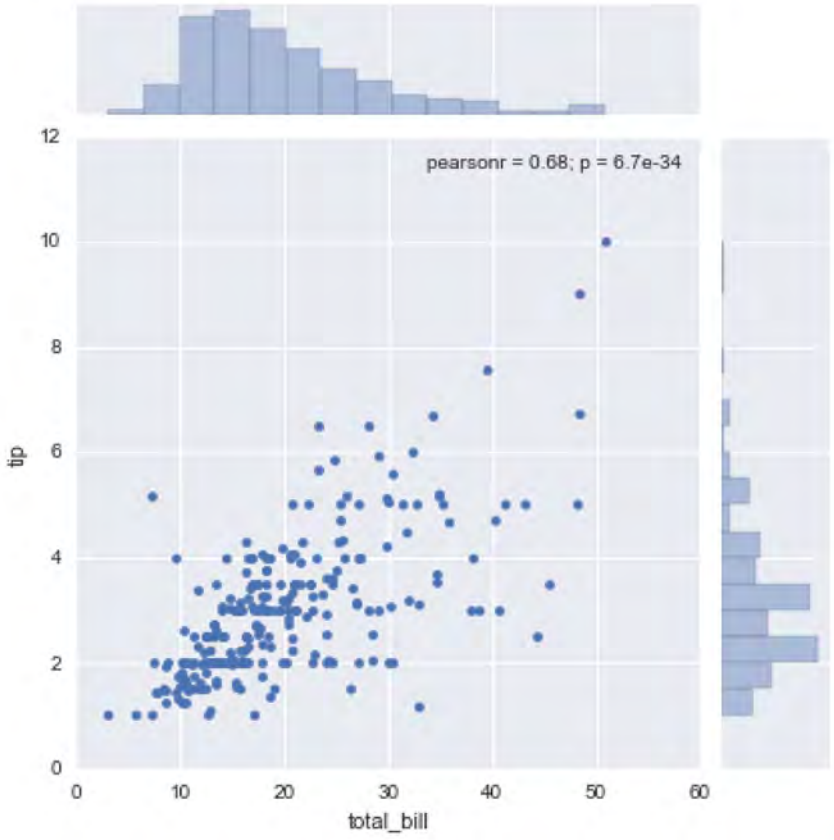
In [6]:
```
sns.jointplot(x='total_bill',y='tip',data=tips,kind='kde')
```

Out[6]:
```
<seaborn.axisgrid.JointGrid at 0x2578893d520>
```



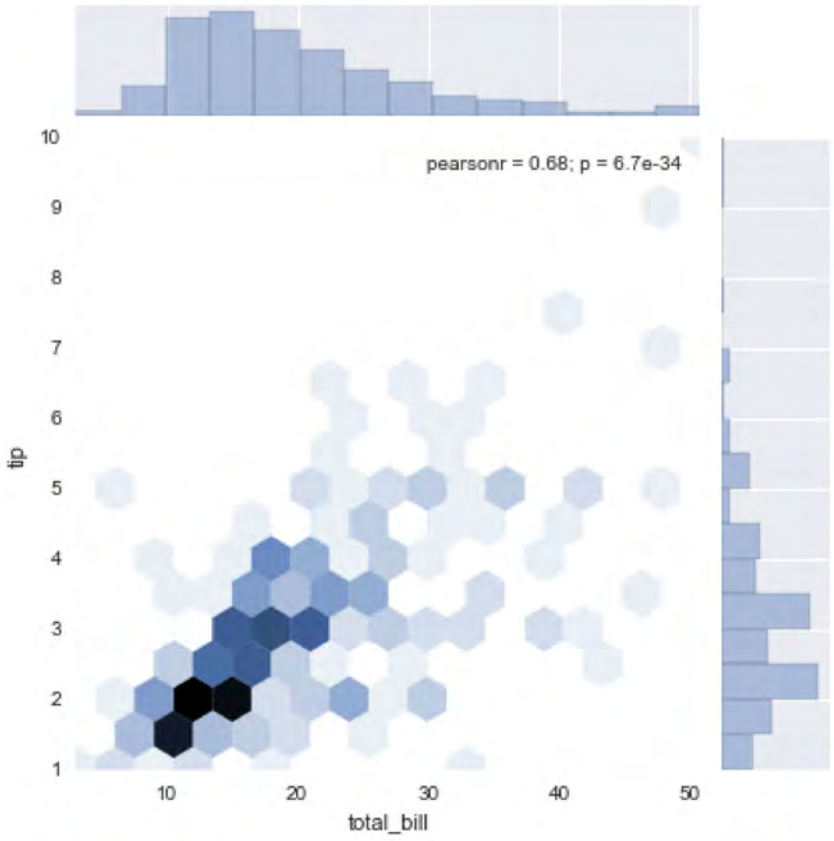
In [12]:
```
sns.jointplot(x='total_bill',y='tip',data=tips,kind='scatter')
```

Out[12]:
```
<seaborn.axisgrid.JointGrid at 0x11cfb28d0>
```



In [15]:
```
sns.jointplot(x='total_bill',y='tip',data=tips,kind='hex')
```

Out[15]:
```
<seaborn.axisgrid.JointGrid at 0x11d96f160>
```

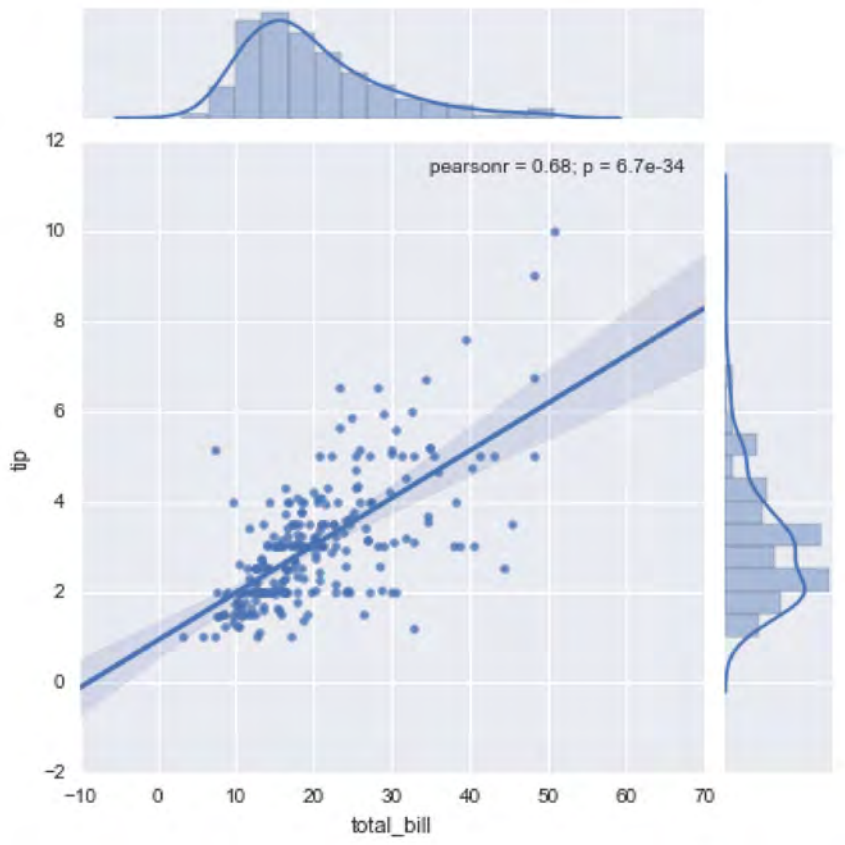


In [17]:
```
sns.jointplot(x='total_bill',y='tip',data=tips,kind='reg')
```

```
/Users/marci/anaconda/lib/python3.5/site-packages/statsmodels/nonparametric/kdetool
s.py:20: VisibleDeprecationWarning: using a non-integer number instead of an integer
```

```
will result in an error in the future
  y = X[:m/2+1] + np.r_[0,X[m/2+1:],0]*1j
```

Out[17]: `<seaborn.axisgrid.JointGrid at 0x11e0cfba8>`



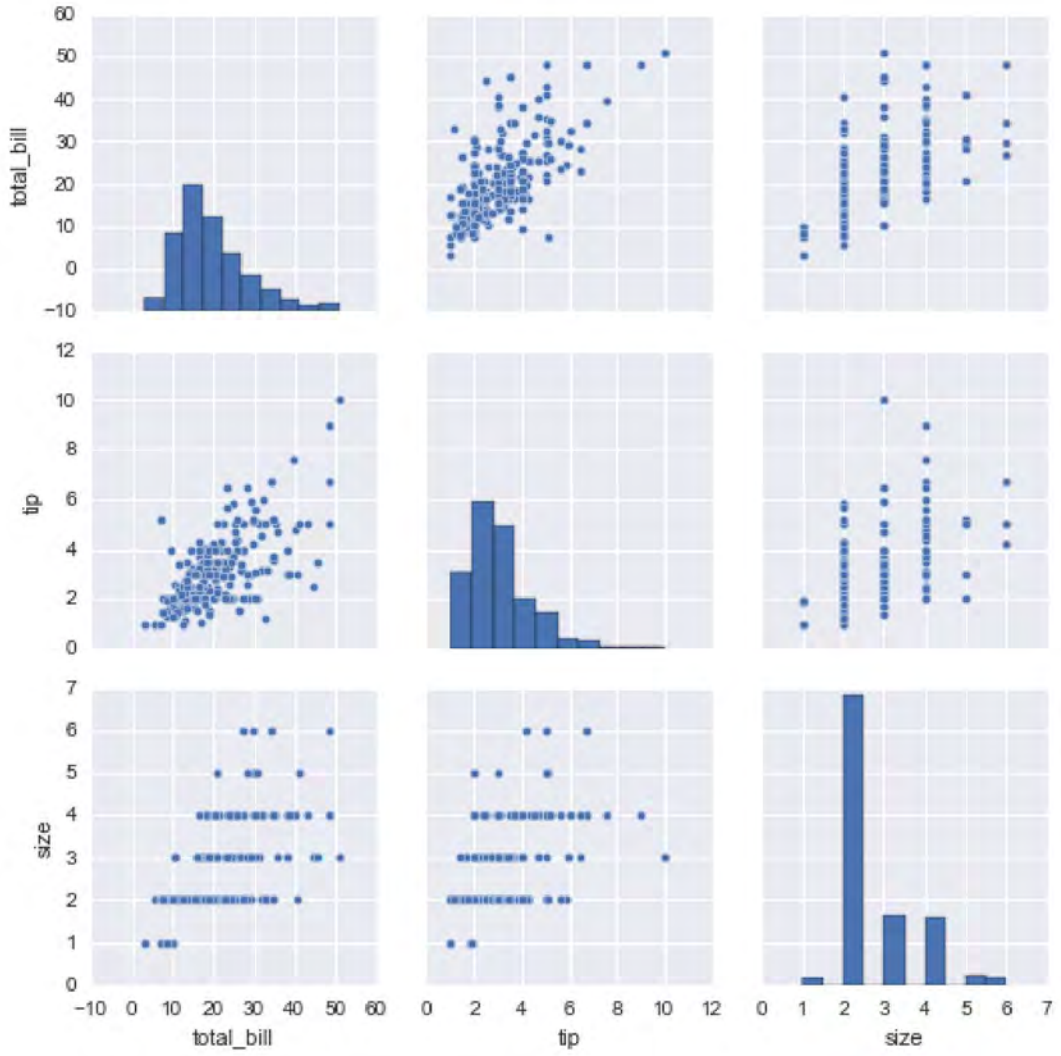
pairplot

pairplot will plot pairwise relationships across an entire dataframe (for the numerical columns) and supports a color hue argument (for categorical columns).

In [ ]:
```
sns.pairplot()
```

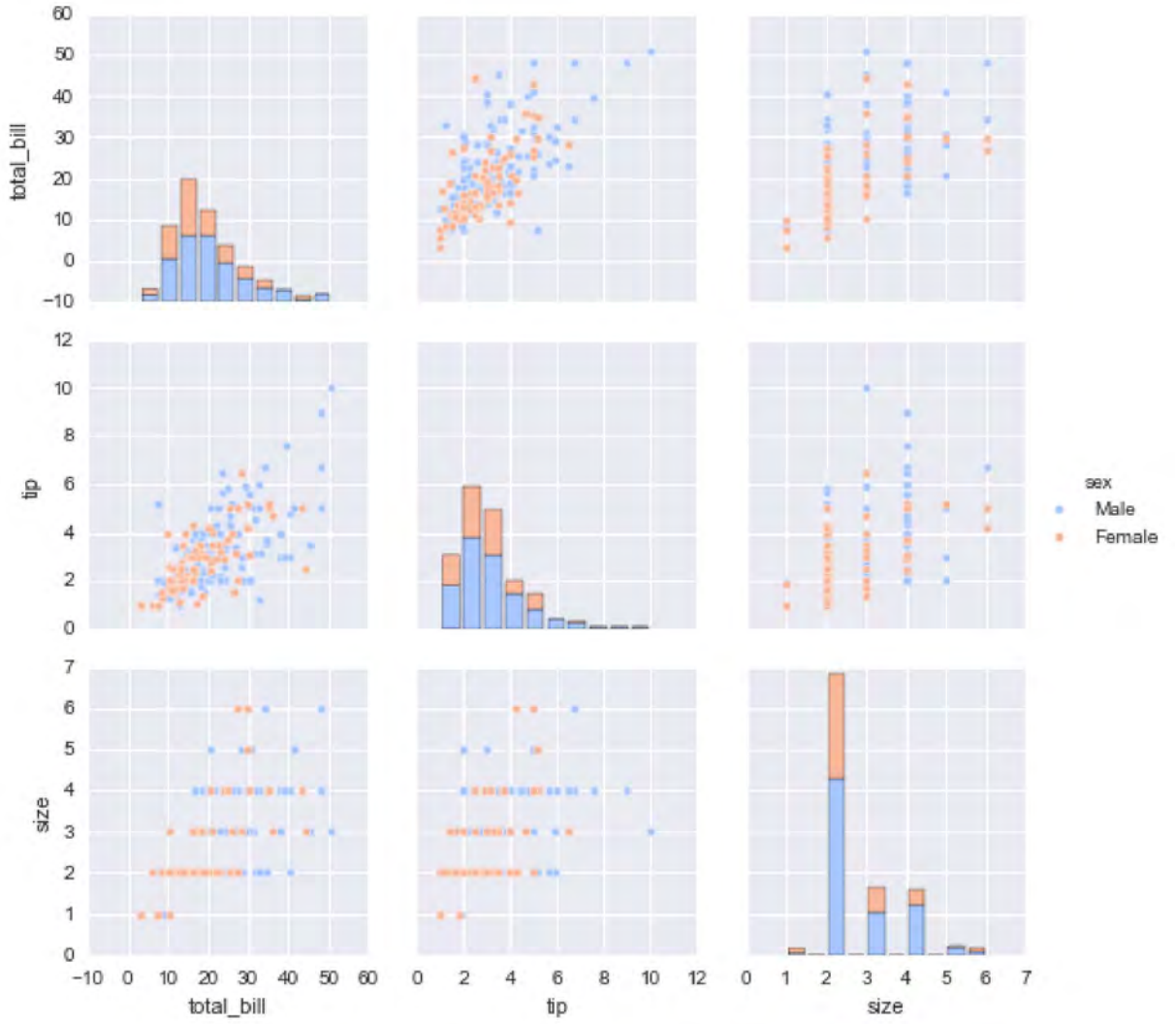
In [18]:
```
sns.pairplot(tips)
```

Out[18]: `<seaborn.axisgrid.PairGrid at 0x11e844208>`



In [21]:
```
sns.pairplot(tips,hue='sex',palette='coolwarm')
```

Out[21]:
```
<seaborn.axisgrid.PairGrid at 0x11ff7a828>
```



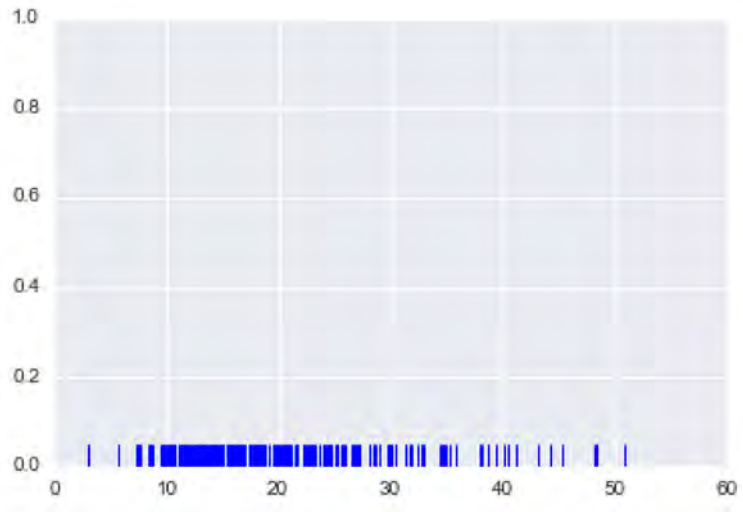
rugplot

rugplots are actually a very simple concept, they just draw a dash mark for every point on a univariate distribution. They are the building block of a KDE plot:

In [ ]:
```
sns.rugplot()
```

In [22]:
```
sns.rugplot(tips['total_bill'])
```

Out[22]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1207c8b70>
```



kdeplot

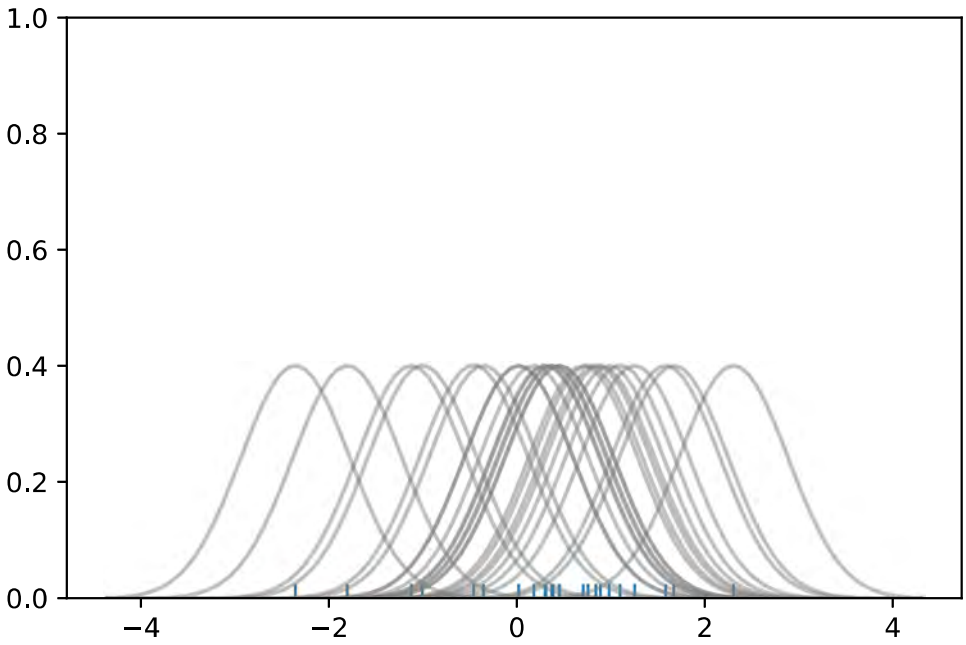
kdeplots are Kernel Density Estimation plots. These KDE plots replace every single observation with a Gaussian (Normal) distribution centered around that value. For example:

In [11]:

```
# Don't worry about understanding this code!
# It's just for the diagram below
import numpy as np
import matplotlib.pyplot as plt
from scipy import stats

#Create dataset
dataset = np.random.randn(25)

# Create another rugplot
sns.rugplot(dataset)

# Set up the x-axis for the plot
x_min = dataset.min() - 2
x_max = dataset.max() + 2

# 100 equally spaced points from x_min to x_max
x_axis = np.linspace(x_min,x_max,100)

# Set up the bandwidth, for info on this:
url = 'http://en.wikipedia.org/wiki/Kernel_density_estimation#Practical_estimation_o

bandwidth = ((4*dataset.std()**5)/(3*len(dataset)))**.2


# Create an empty kernel list
kernel_list = []

# Plot each basis function
for data_point in dataset:

    # Create a kernel for each point and append to list
    kernel = stats.norm(data_point,bandwidth).pdf(x_axis)
    kernel_list.append(kernel)

    #Scale for plotting
    kernel = kernel / kernel.max()
    kernel = kernel * .4
    plt.plot(x_axis,kernel,color = 'grey',alpha=0.5)

plt.ylim(0,1)
```

Out[11]:

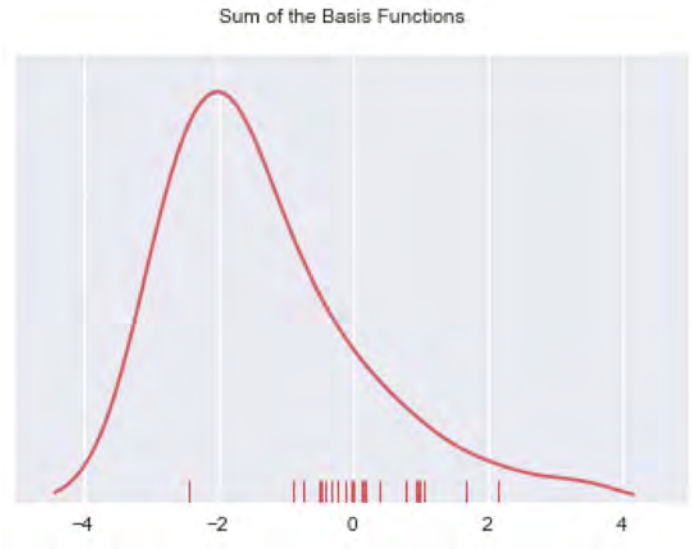
```
(0.0, 1.0)
```



In [37]:

```
# To get the kde plot we can sum these basis functions.

# Plot the sum of the basis function
sum_of_kde = np.sum(kernel_list,axis=0)

# Plot figure
fig = plt.plot(x_axis,sum_of_kde,color='indianred')

# Add the initial rugplot
sns.rugplot(dataset,c = 'indianred')

# Get rid of y-tick marks
plt.yticks([])

# Set title
plt.suptitle("Sum of the Basis Functions")
```

Out[37]: <matplotlib.text.Text at 0x121c41da0>



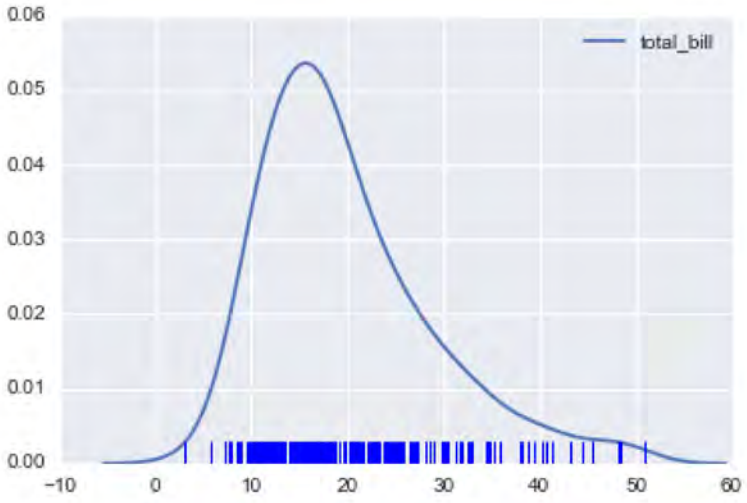
So with our tips dataset:

In [41]:

```
sns.kdeplot(tips['total_bill'])
sns.rugplot(tips['total_bill'])
```

```
/Users/marci/anaconda/lib/python3.5/site-packages/statsmodels/nonparametric/kdetool
s.py:20: VisibleDeprecationWarning: using a non-integer number instead of an integer
will result in an error in the future
  y = X[:m/2+1] + np.r_[0,X[m/2+1:],0]*1j
```

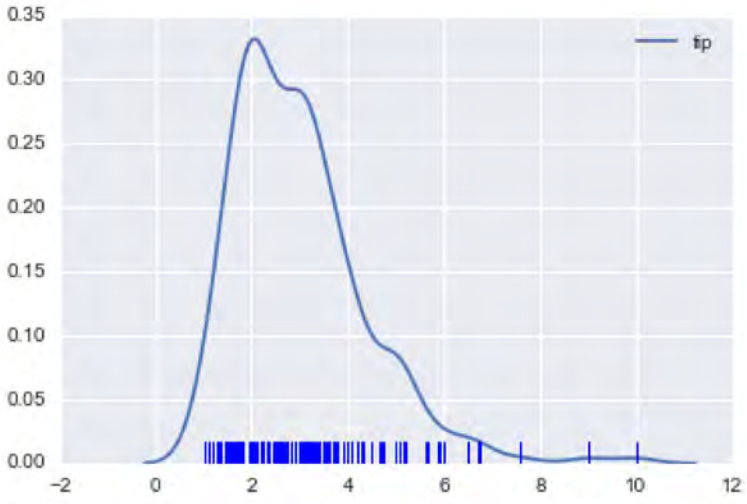
Out[41]: `<matplotlib.axes._subplots.AxesSubplot at 0x121b82c50>`



In [42]:
```
sns.kdeplot(tips['tip'])
sns.rugplot(tips['tip'])
```

```
/Users/marci/anaconda/lib/python3.5/site-packages/statsmodels/nonparametric/kdetool
s.py:20: VisibleDeprecationWarning: using a non-integer number instead of an integer
will result in an error in the future
  y = X[:m/2+1] + np.r_[0,X[m/2+1:],0]*1j
```

Out[42]: `<matplotlib.axes._subplots.AxesSubplot at 0x12252cfd0>`



In [ ]:
```
sns.kdeplot()
```

Categorical Data Plots

Now let's discuss using seaborn to plot categorical data! There are a few main plot types for this:

- factorplot
- boxplot
- violinplot
- stripplot
- swarmplot

- barplot
- countplot

Let's go through examples of each!

In [1]:
```
import seaborn as sns
import numpy as np
%matplotlib inline
```

In [2]:
```
tips = sns.load_dataset('tips')
tips.head()
```

Out[2]:

	total_bill	tip	sex	smoker	day	time	size
0	16.99	1.01	Female	No	Sun	Dinner	2
1	10.34	1.66	Male	No	Sun	Dinner	3
2	21.01	3.50	Male	No	Sun	Dinner	3
3	23.68	3.31	Male	No	Sun	Dinner	2
4	24.59	3.61	Female	No	Sun	Dinner	4

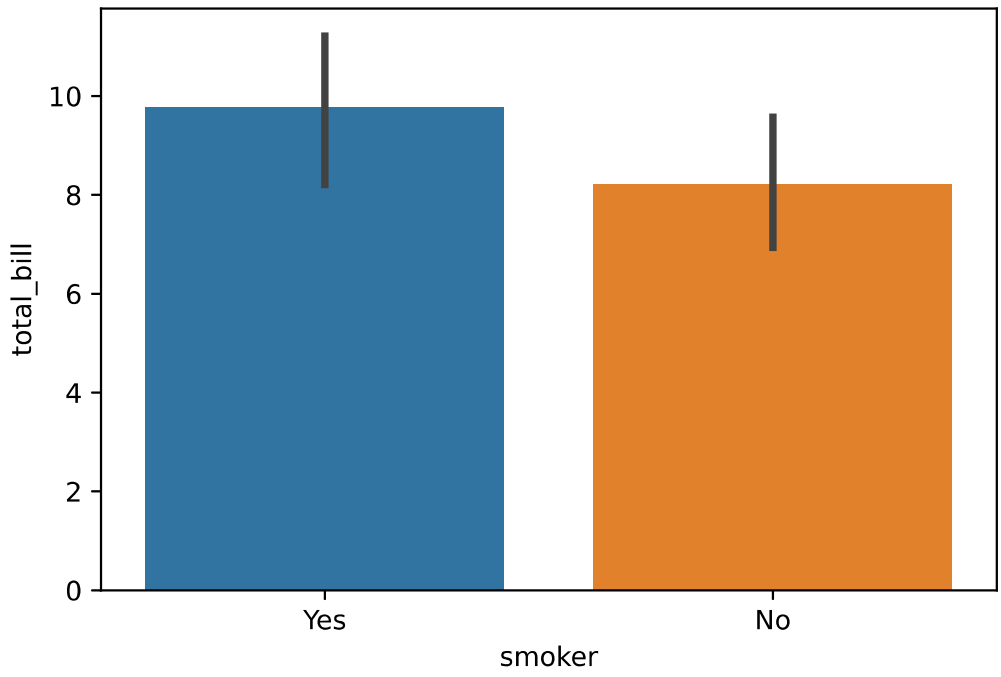
barplot and countplot

These very similar plots allow you to get aggregate data off a categorical feature in your data. **barplot** is a general plot that allows you to aggregate the categorical data based off some function, by default the mean:

In [ ]:
```
sns.barplot()
```

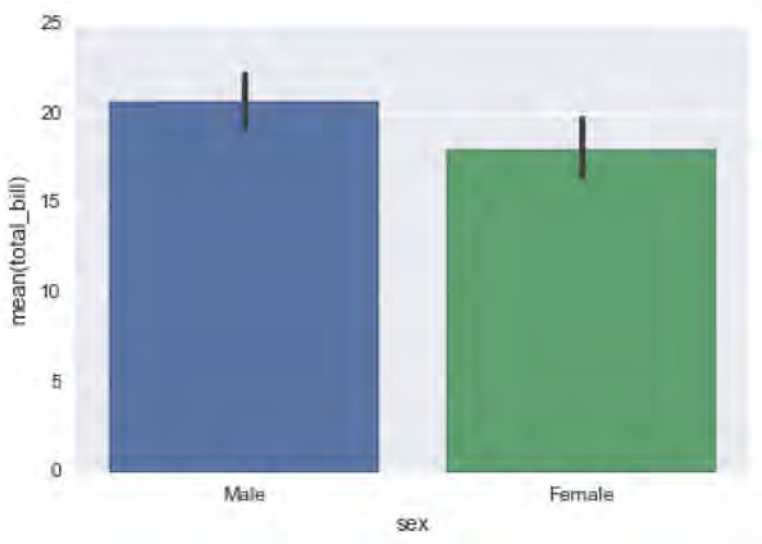
In [6]:
```
sns.barplot()
```

Out[6]:
```
<AxesSubplot:xlabel='smoker', ylabel='total_bill'>
```



In [8]:
```
sns.barplot(x='sex',y='total_bill',data=tips)
```

Out[8]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x11c99b8d0>
```

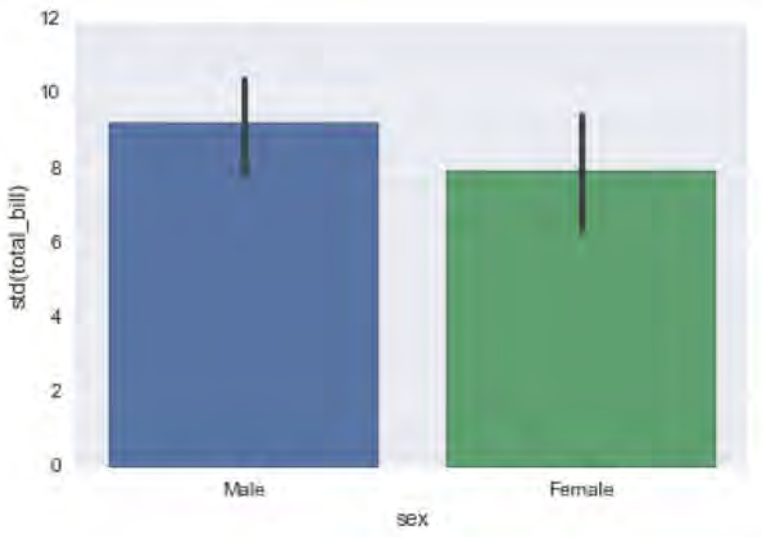


In [10]:
```
import numpy as np
```

You can change the estimator object to your own function, that converts a vector to a scalar:

In [11]:
```
sns.barplot(x='sex',y='total_bill',data=tips,estimator=np.std)
```

Out[11]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x11c9b00b8>
```



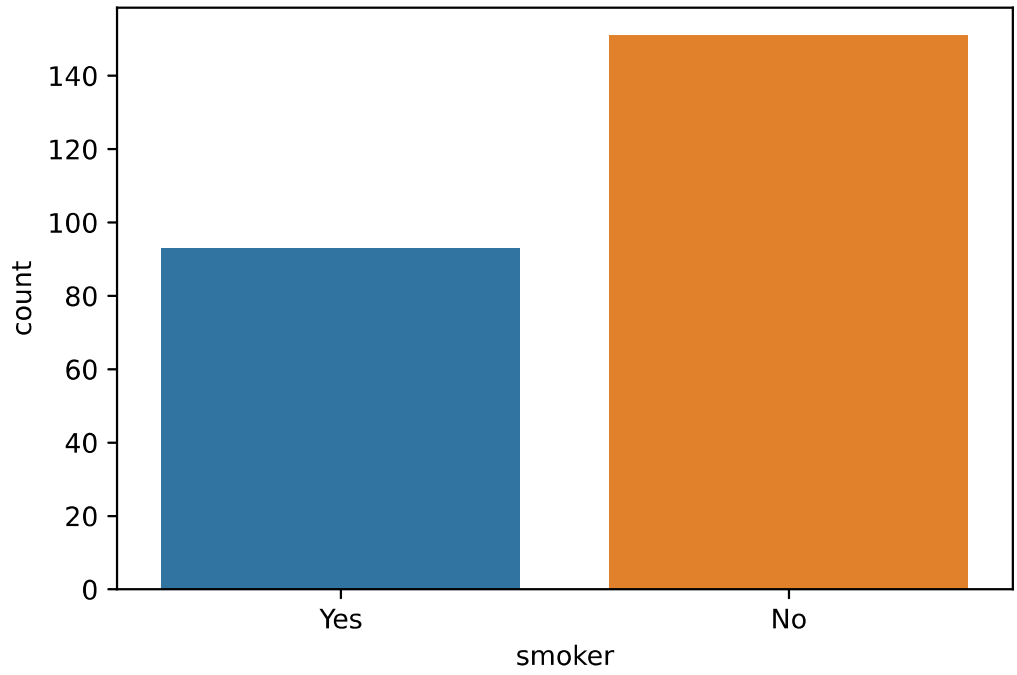
countplot

This is essentially the same as barplot except the estimator is explicitly counting the number of occurrences. Which is why we only pass the x value:

In [ ]:
```
sns.countplot()
```

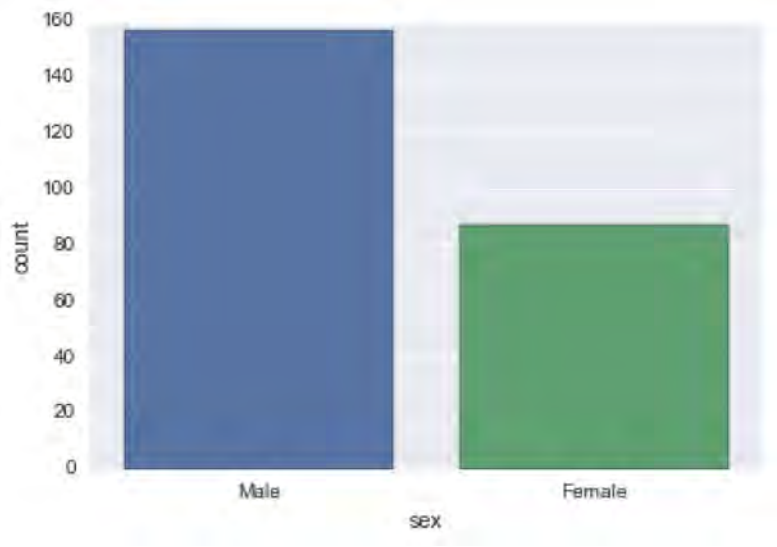
In [7]:
```
sns.countplot(x='smoker',data=tips)
```

Out[7]:
```
<AxesSubplot:xlabel='smoker', ylabel='count'>
```



In [13]:
```
sns.countplot(x='sex',data=tips)
```

Out[13]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1153276d8>
```



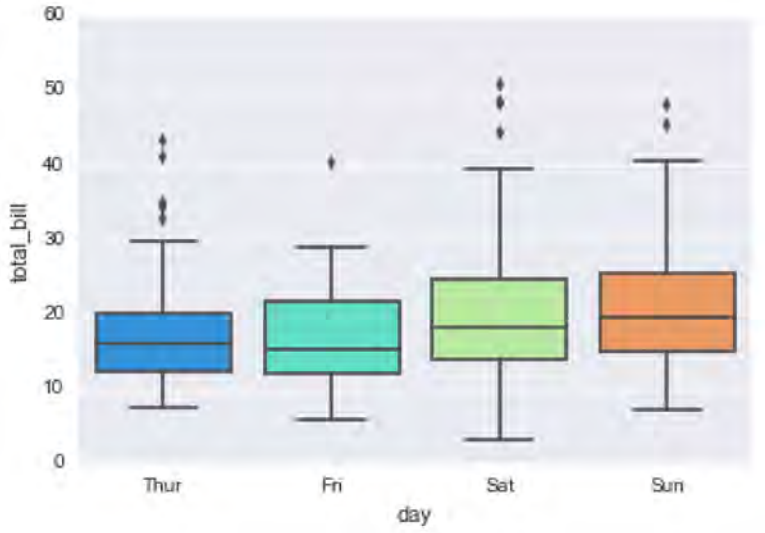
boxplot and violinplot

boxplots and violinplots are used to shown the distribution of categorical data. A box plot (or box-and-whisker plot) shows the distribution of quantitative data in a way that facilitates comparisons between variables or across levels of a categorical variable. The box shows the quartiles of the dataset while the whiskers extend to show the rest of the distribution, except for points that are determined to be “outliers” using a method that is a function of the inter-quartile range.

In [ ]:
```
sns.boxplot()
```

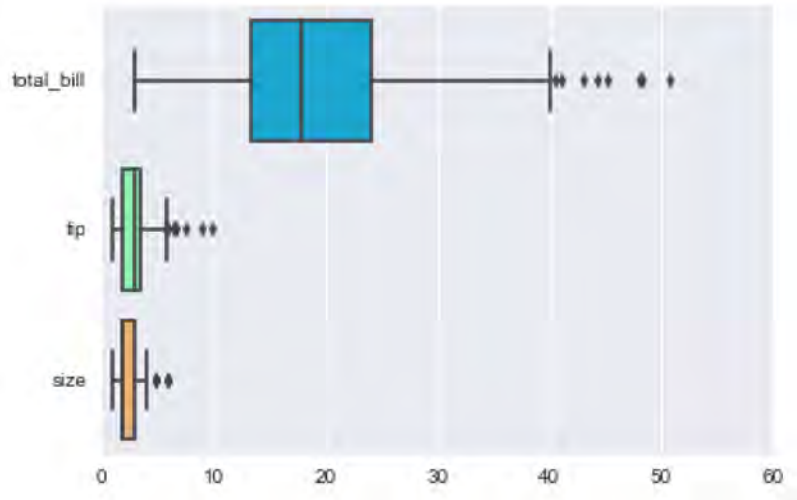
In [22]:
```
sns.boxplot(x="day", y="total_bill", data=tips,palette='rainbow')
```

Out[22]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x11db81630>
```



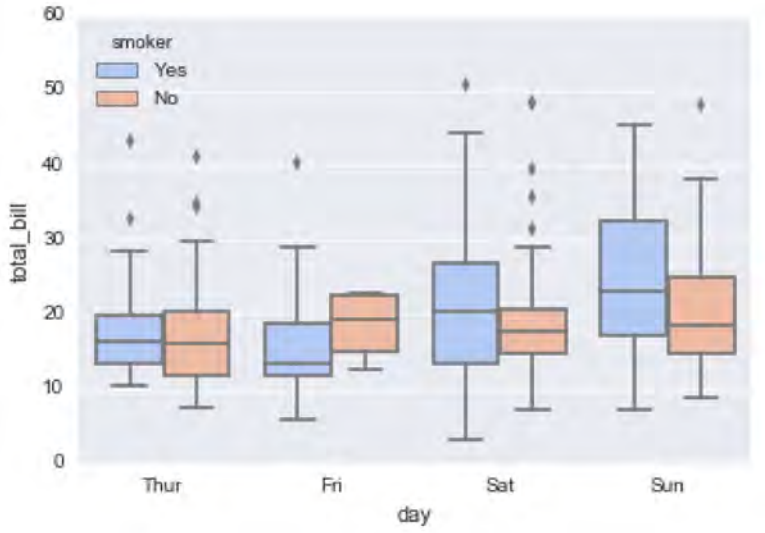
In [25]:
```
# Can do entire dataframe with orient='h'
sns.boxplot(data=tips,palette='rainbow',orient='h')
```

Out[25]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x11e2c0b00>
```



In [26]:
```
sns.boxplot(x="day", y="total_bill", hue="smoker",data=tips, palette="coolwarm")
```

Out[26]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x11e2c77f0>
```



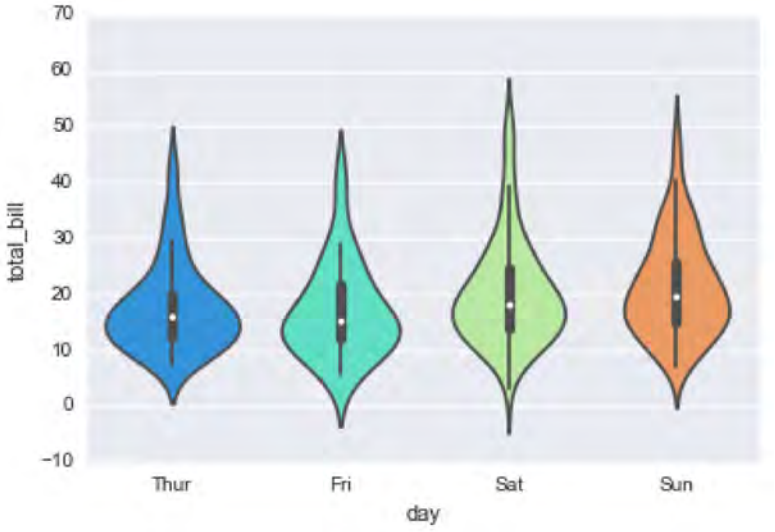
violinplot

A violin plot plays a similar role as a box and whisker plot. It shows the distribution of quantitative data across several levels of one (or more) categorical variables such that those distributions can be compared. Unlike a box plot, in which all of the plot components correspond to actual datapoints, the violin plot features a kernel density estimation of the underlying distribution.

In [ ]:
```
sns.violinplot()
```

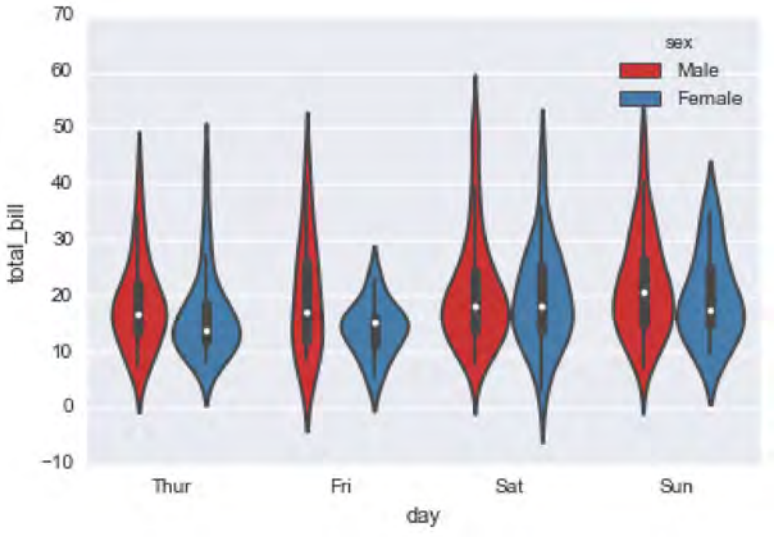
In [27]:
```
sns.violinplot(x="day", y="total_bill", data=tips,palette='rainbow')
```

Out[27]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x11e682ba8>
```



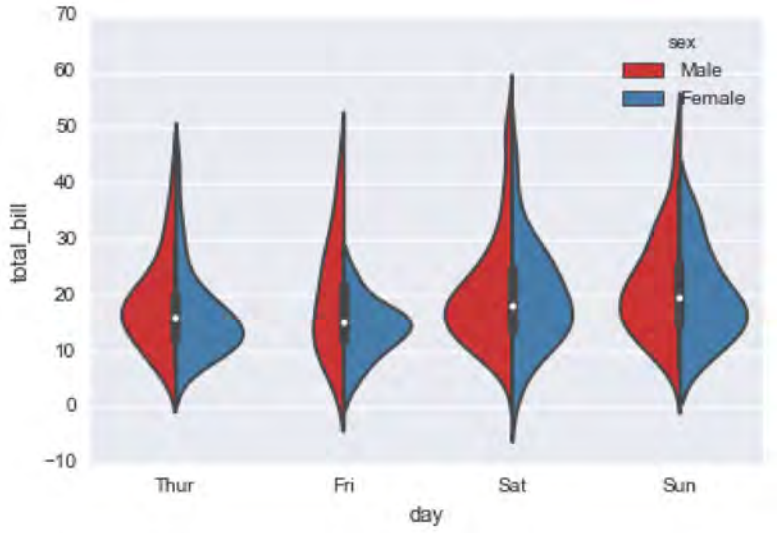
In [37]:
```
sns.violinplot(x="day", y="total_bill", data=tips,hue='sex',palette='Set1')
```

Out[37]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x11f739dd8>
```



In [36]:
```
sns.violinplot(x="day", y="total_bill", data=tips,hue='sex',split=True,palette='Set1
```

Out[36]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x11f4d0710>
```



stripplot and swarmplot

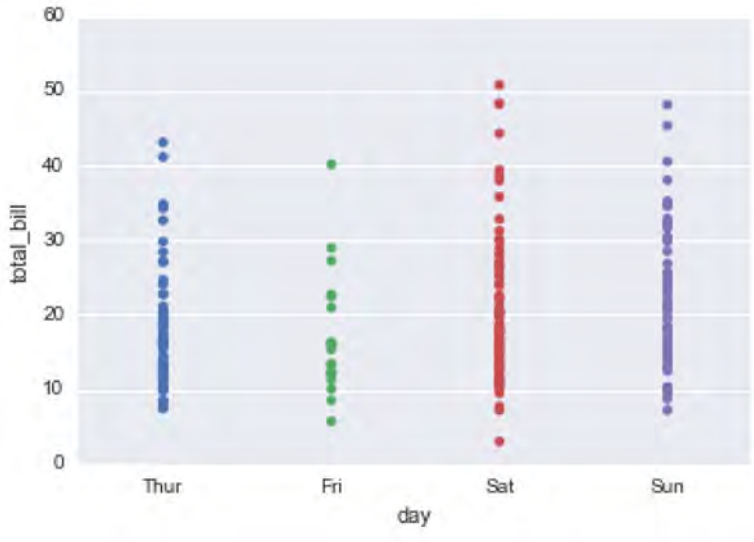
The stripplot will draw a scatterplot where one variable is categorical. A strip plot can be drawn on its own, but it is also a good complement to a box or violin plot in cases where you want to show all observations along with some representation of the underlying distribution.

The swarmplot is similar to stripplot(), but the points are adjusted (only along the categorical axis) so that they don't overlap. This gives a better representation of the distribution of values, although it does not scale as well to large numbers of observations (both in terms of the ability to show all the points and in terms of the computation needed to arrange them).

In [ ]:
```
sns.stripplot
```

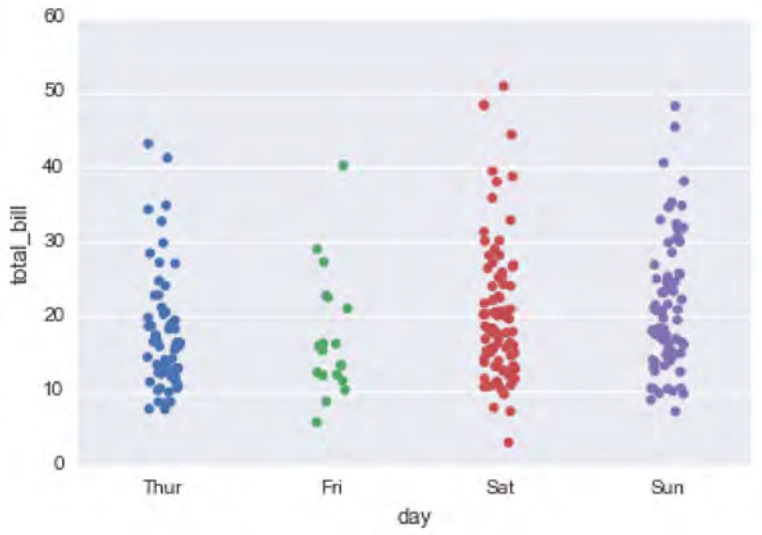
In [38]:
```
sns.stripplot(x="day", y="total_bill", data=tips)
```

Out[38]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x120272278>
```



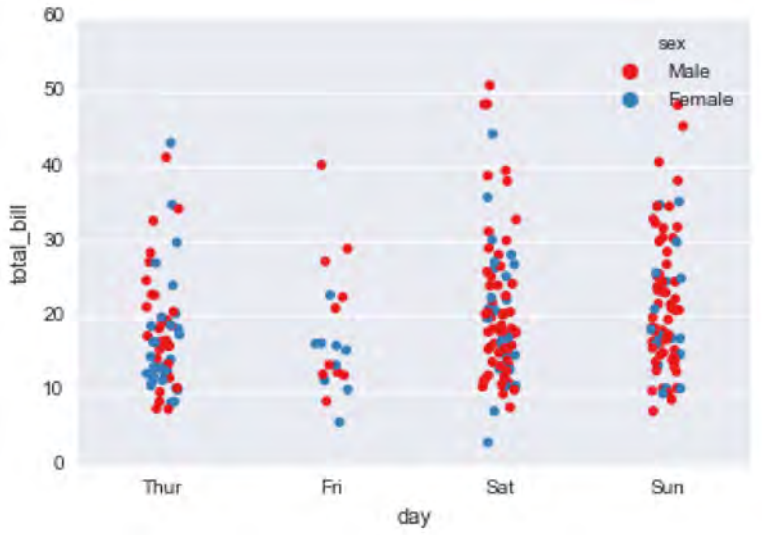
In [39]:
```
sns.stripplot(x="day", y="total_bill", data=tips,jitter=True)
```

Out[39]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1203a8470>
```



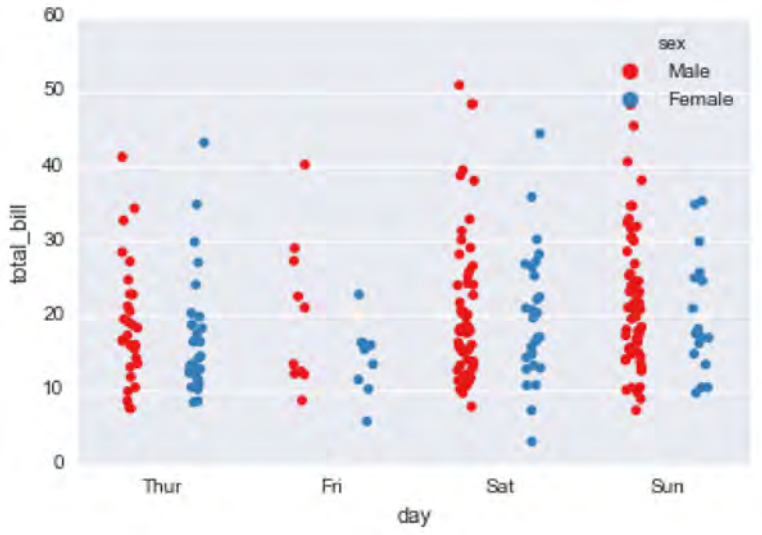
In [42]:
```
sns.stripplot(x="day", y="total_bill", data=tips,jitter=True,hue='sex',palette='Set1
```

Out[42]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x12092e518>
```



In [43]:
```
sns.stripplot(x="day", y="total_bill", data=tips,jitter=True,hue='sex',palette='Set1
```

Out[43]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x12099db70>
```

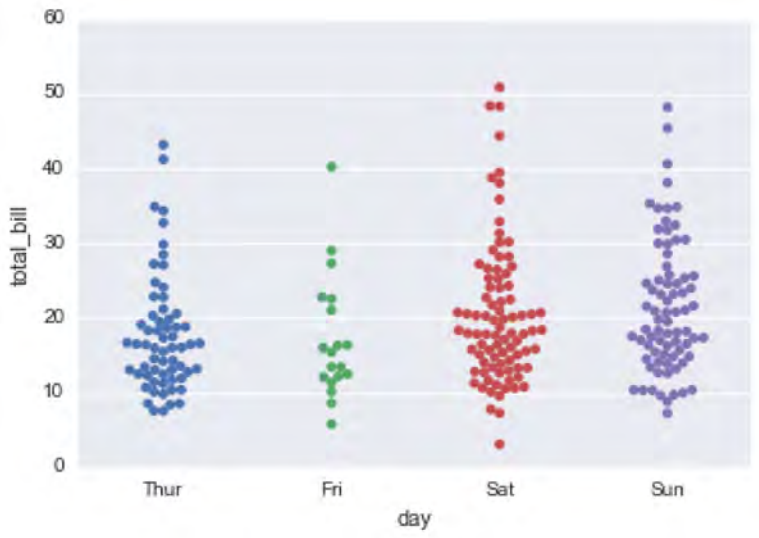


In [ ]:

```
sns.swarmplot()
```

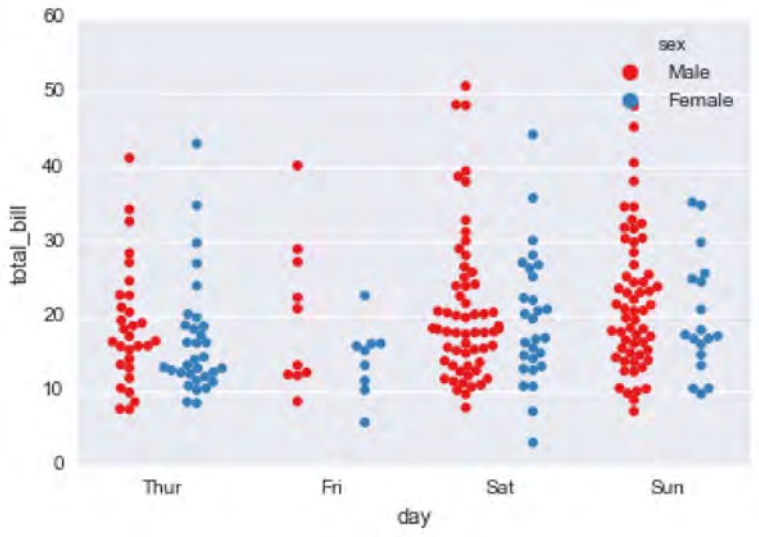
In [44]:
```
sns.swarmplot(x="day", y="total_bill", data=tips)
```

Out[44]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x120c463c8>
```



In [47]:
```
sns.swarmplot(x="day", y="total_bill",hue='sex',data=tips, palette="Set1", split=Tru
```

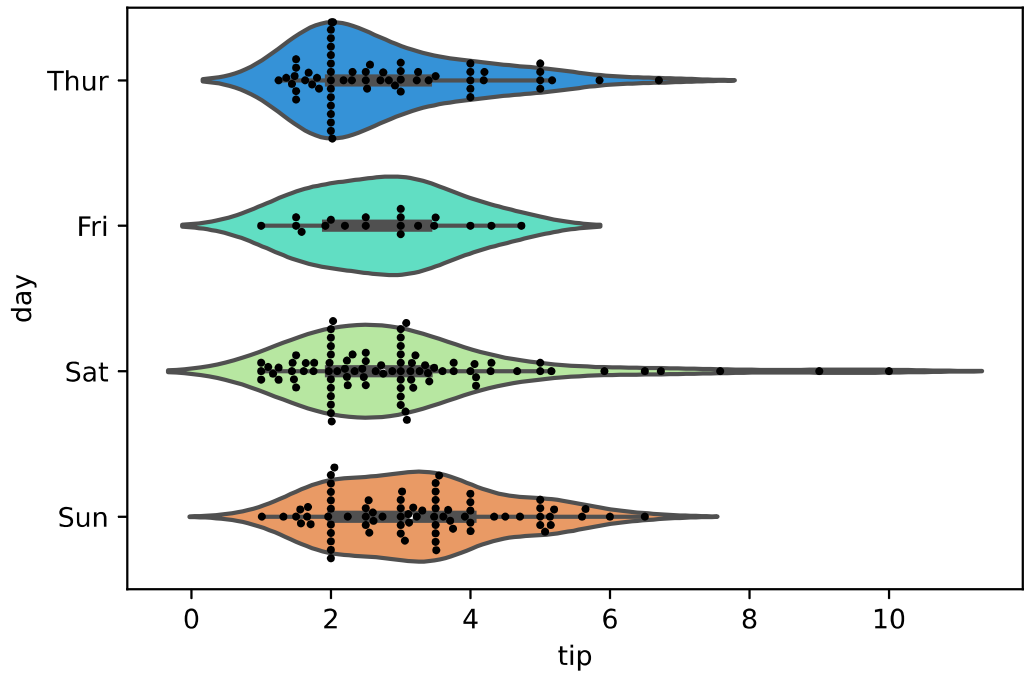
Out[47]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1211b6da0>
```



Combining Categorical Plots

In [9]:
```
sns.violinplot(x="tip", y="day", data=tips,palette='rainbow')
sns.swarmplot(x="tip", y="day", data=tips,color = "black", size=3)
```

Out[9]:
```
<AxesSubplot:xlabel='tip', ylabel='day'>
```



factorplot

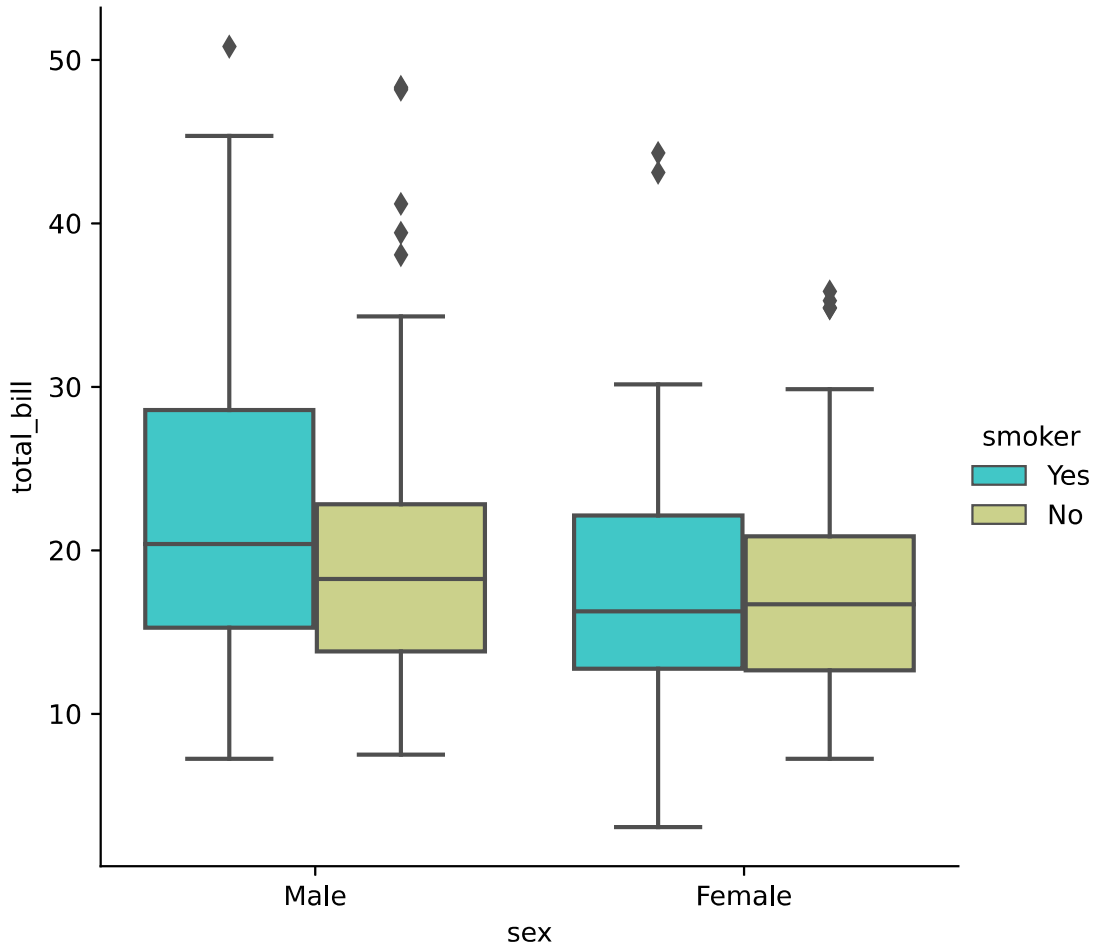
factorplot is the most general form of a categorical plot. It can take in a **kind** parameter to adjust the plot type:

In [ ]:
```
sns.catplot()
```

In [18]:
```
sns.factorplot(x='sex',y='total_bill', hue="smoker", palette='rainbow' ,data=tips,ki
```

```
C:\Users\HickUp\anaconda3\envs\MyPyt39\lib\site-packages\seaborn\categorical.py:371
4: UserWarning: The `factorplot` function has been renamed to `catplot`. The origina
l name will be removed in a future release. Please update your code. Note that the d
efault `kind` in `factorplot` (``'point'``) has changed ``'strip'`` in `catplot`.
  warnings.warn(msg)
```

Out[18]:
```
<seaborn.axisgrid.FacetGrid at 0x2c0bb70df10>
```



Matrix Plots

Matrix plots allow you to plot data as color-encoded matrices and can also be used to indicate clusters within the data (later in the machine learning section we will learn how to formally cluster data).

Let's begin by exploring seaborn's heatmap and clutermap:

In [1]:
```
import seaborn as sns
%matplotlib inline
```

In [2]:
```
flights = sns.load_dataset('flights')
```

In [3]:
```
tips = sns.load_dataset('tips')
```

In [11]:
```
tips.head()
```

Out[11]:

	total_bill	tip	sex	smoker	day	time	size
0	16.99	1.01	Female	No	Sun	Dinner	2
1	10.34	1.66	Male	No	Sun	Dinner	3
2	21.01	3.50	Male	No	Sun	Dinner	3
3	23.68	3.31	Male	No	Sun	Dinner	2
4	24.59	3.61	Female	No	Sun	Dinner	4

In [4]:
```
flights.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 144 entries, 0 to 143
Data columns (total 3 columns):
 #   Column      Non-Null Count  Dtype   
---  ------      --------------  -----   
 0   year        144 non-null    int64   
 1   month       144 non-null    category
 2   passengers  144 non-null    int64   
dtypes: category(1), int64(2)
memory usage: 2.9 KB
```

In [4]:
```
flights.head()
```

Out[4]:

	year	month	passengers
0	1949	January	112
1	1949	February	118
2	1949	March	132
3	1949	April	129
4	1949	May	121

Heatmap

In order for a heatmap to work properly, your data should already be in a matrix form, the sns.heatmap function basically just colors it in for you. For example:

In [12]:
```
tips.head()
```

Out[12]:

	total_bill	tip	sex	smoker	day	time	size
0	16.99	1.01	Female	No	Sun	Dinner	2
1	10.34	1.66	Male	No	Sun	Dinner	3
2	21.01	3.50	Male	No	Sun	Dinner	3
3	23.68	3.31	Male	No	Sun	Dinner	2
4	24.59	3.61	Female	No	Sun	Dinner	4

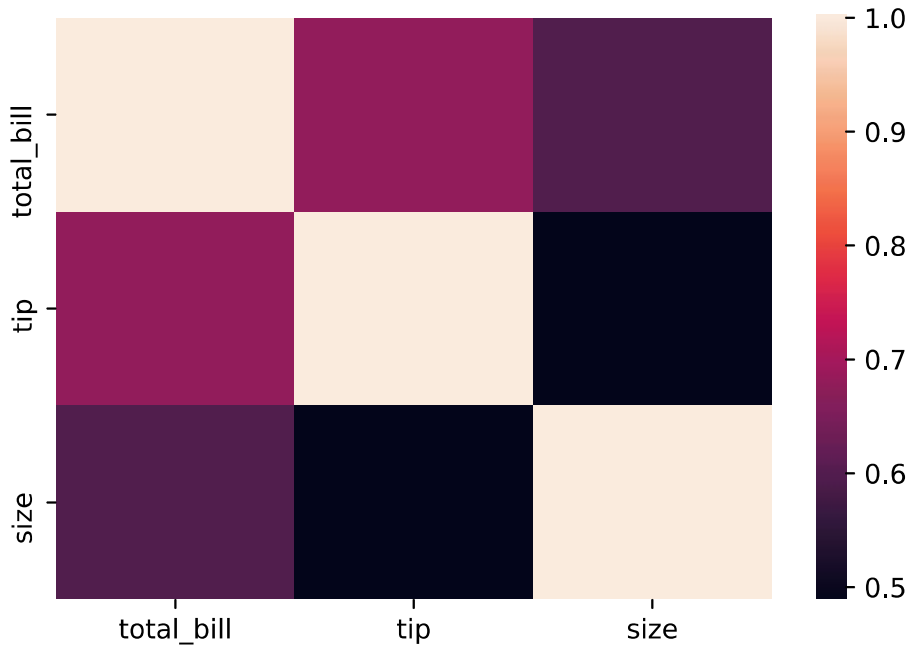
In [14]:
```
# Matrix form for correlation data
tips.corr()
```

Out[14]:

	total_bill	tip	size
total_bill	1.000000	0.675734	0.598315
tip	0.675734	1.000000	0.489299
size	0.598315	0.489299	1.000000

In [10]:
```
sns.heatmap(tips.corr())
```

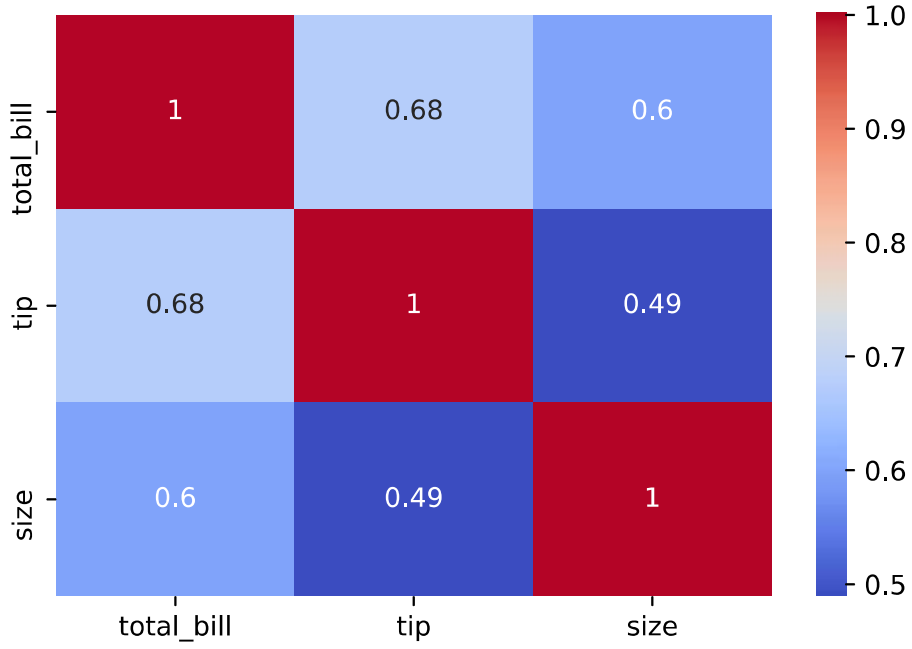
Out[10]: <AxesSubplot:>



In [ ]:
```
sns.heatmap()
```

In [9]:
```
sns.heatmap(tips.corr(),cmap='coolwarm',annot=True)
```

Out[9]: <AxesSubplot:>



In [ ]:
```

```

Or for the flights data:

In [6]:
```
flights.pivot_table(values='passengers',index='month',columns='year')
```

Out[6]:

year	1949	1950	1951	1952	1953	1954	1955	1956	1957	1958	1959	1960

month	1949	1950	1951	1952	1953	1954	1955	1956	1957	1958	1959	1960
Jan	112	115	145	171	196	204	242	284	315	340	360	417
Feb	118	126	150	180	196	188	233	277	301	318	342	391
Mar	132	141	178	193	236	235	267	317	356	362	406	419
Apr	129	135	163	181	235	227	269	313	348	348	396	461
May	121	125	172	183	229	234	270	318	355	363	420	472
Jun	135	149	178	218	243	264	315	374	422	435	472	535
Jul	148	170	199	230	264	302	364	413	465	491	548	622
Aug	148	170	199	242	272	293	347	405	467	505	559	606
Sep	136	158	184	209	237	259	312	355	404	404	463	508
Oct	119	133	162	191	211	229	274	306	347	359	407	461
Nov	104	114	146	172	180	203	237	271	305	310	362	390
Dec	118	140	166	194	201	229	278	306	336	337	405	432

In [7]:
```
pvflights = flights.pivot_table(values='passengers',index='month',columns='year')
sns.heatmap(pvflights)
```

Out[7]: <AxesSubplot:xlabel='year', ylabel='month'>

Beginner's Python Cheat Sheet

Variables and Strings

Variables are used to store values. A string is a series of characters, surrounded by single or double quotes.

Hello world

```
print("Hello world!")
```

Hello world with a variable

```
msg = "Hello world!"
print(msg)
```

Concatenation (combining strings)

```
first_name = 'albert'
last_name = 'einstein'
full_name = first_name + ' ' + last_name
print(full_name)
```

Lists

A list stores a series of items in a particular order. You access items using an index, or within a loop.

Make a list

```
bikes = ['trek', 'redline', 'giant']
```

Get the first item in a list

```
first_bike = bikes[0]
```

Get the last item in a list

```
last_bike = bikes[-1]
```

Looping through a list

```
for bike in bikes:
    print(bike)
```

Adding items to a list

```
bikes = []
bikes.append('trek')
bikes.append('redline')
bikes.append('giant')
```

Making numerical lists

```
squares = []
for x in range(1, 11):
    squares.append(x**2)
```

Lists (cont.)

List comprehensions

```
squares = [x**2 for x in range(1, 11)]
```

Slicing a list

```
finishers = ['sam', 'bob', 'ada', 'bea']
first_two = finishers[:2]
```

Copying a list

```
copy_of_bikes = bikes[:]
```

Tuples

Tuples are similar to lists, but the items in a tuple can't be modified.

Making a tuple

```
dimensions = (1920, 1080)
```

If statements

If statements are used to test for particular conditions and respond appropriately.

Conditional tests

```
equals            x == 42
not equal         x != 42
greater than      x > 42
  or equal to     x >= 42
less than         x < 42
  or equal to     x <= 42
```

Conditional test with lists

```
'trek' in bikes
'surly' not in bikes
```

Assigning boolean values

```
game_active = True
can_edit = False
```

A simple if test

```
if age >= 18:
    print("You can vote!")
```

If-elif-else statements

```
if age < 4:
    ticket_price = 0
elif age < 18:
    ticket_price = 10
else:
    ticket_price = 15
```

Dictionaries

Dictionaries store connections between pieces of information. Each item in a dictionary is a key-value pair.

A simple dictionary

```
alien = {'color': 'green', 'points': 5}
```

Accessing a value

```
print("The alien's color is " + alien['color'])
```

Adding a new key-value pair

```
alien['x_position'] = 0
```

Looping through all key-value pairs

```
fav_numbers = {'eric': 17, 'ever': 4}
for name, number in fav_numbers.items():
    print(name + ' loves ' + str(number))
```

Looping through all keys

```
fav_numbers = {'eric': 17, 'ever': 4}
for name in fav_numbers.keys():
    print(name + ' loves a number')
```

Looping through all the values

```
fav_numbers = {'eric': 17, 'ever': 4}
for number in fav_numbers.values():
    print(str(number) + ' is a favorite')
```

User input

Your programs can prompt the user for input. All input is stored as a string.

Prompting for a value

```
name = input("What's your name? ")
print("Hello, " + name + "!")
```

Prompting for numerical input

```
age = input("How old are you? ")
age = int(age)

pi = input("What's the value of pi? ")
pi = float(pi)
```

While loops

A while loop repeats a block of code as long as a certain condition is true.

A simple while loop

```
current_value = 1
while current_value <= 5:
    print(current_value)
    current_value += 1
```

Letting the user choose when to quit

```
msg = ''
while msg != 'quit':
    msg = input("What's your message? ")
    print(msg)
```

Functions

Functions are named blocks of code, designed to do one specific job. Information passed to a function is called an argument, and information received by a function is called a parameter.

A simple function

```
def greet_user():
    """Display a simple greeting."""
    print("Hello!")

greet_user()
```

Passing an argument

```
def greet_user(username):
    """Display a personalized greeting."""
    print("Hello, " + username + "!")

greet_user('jesse')
```

Default values for parameters

```
def make_pizza(topping='bacon'):
    """Make a single-topping pizza."""
    print("Have a " + topping + " pizza!")

make_pizza()
make_pizza('pepperoni')
```

Returning a value

```
def add_numbers(x, y):
    """Add two numbers and return the sum."""
    return x + y

sum = add_numbers(3, 5)
print(sum)
```

Classes

A class defines the behavior of an object and the kind of information an object can store. The information in a class is stored in attributes, and functions that belong to a class are called methods. A child class inherits the attributes and methods from its parent class.

Creating a dog class

```
class Dog():
    """Represent a dog."""

    def __init__(self, name):
        """Initialize dog object."""
        self.name = name

    def sit(self):
        """Simulate sitting."""
        print(self.name + " is sitting.")

my_dog = Dog('Peso')

print(my_dog.name + " is a great dog!")
my_dog.sit()
```

Inheritance

```
class SARDog(Dog):
    """Represent a search dog."""

    def __init__(self, name):
        """Initialize the sardog."""
        super().__init__(name)

    def search(self):
        """Simulate searching."""
        print(self.name + " is searching.")

my_dog = SARDog('Willie')

print(my_dog.name + " is a search dog.")
my_dog.sit()
my_dog.search()
```

Infinite Skills

If you had infinite programming skills, what would you build?

As you're learning to program, it's helpful to think about the real-world projects you'd like to create. It's a good habit to keep an "ideas" notebook that you can refer to whenever you want to start a new project. If you haven't done so already, take a few minutes and describe three projects you'd like to create.

Working with files

Your programs can read from files and write to files. Files are opened in read mode ('r') by default, but can also be opened in write mode ('w') and append mode ('a').

Reading a file and storing its lines

```
filename = 'siddhartha.txt'
with open(filename) as file_object:
    lines = file_object.readlines()

for line in lines:
    print(line)
```

Writing to a file

```
filename = 'journal.txt'
with open(filename, 'w') as file_object:
    file_object.write("I love programming.")
```

Appending to a file

```
filename = 'journal.txt'
with open(filename, 'a') as file_object:
    file_object.write("\nI love making games.")
```

Exceptions

Exceptions help you respond appropriately to errors that are likely to occur. You place code that might cause an error in the try block. Code that should run in response to an error goes in the except block. Code that should run only if the try block was successful goes in the else block.

Catching an exception

```
prompt = "How many tickets do you need? "
num_tickets = input(prompt)

try:
    num_tickets = int(num_tickets)
except ValueError:
    print("Please try again.")
else:
    print("Your tickets are printing.")
```

Zen of Python

Simple is better than complex

If you have a choice between a simple and a complex solution, and both work, use the simple solution. Your code will be easier to maintain, and it will be easier for you and others to build on that code later on.

More cheat sheets available at
ehmatthes.github.io/pcc/

Beginner's Python Cheat Sheet - Lists

What are lists?

A list stores a series of items in a particular order. Lists allow you to store sets of information in one place, whether you have just a few items or millions of items. Lists are one of Python's most powerful features readily accessible to new programmers, and they tie together many important concepts in programming.

Defining a list

Use square brackets to define a list, and use commas to separate individual items in the list. Use plural names for lists, to make your code easier to read.

Making a list

```
users = ['val', 'bob', 'mia', 'ron', 'ned']
```

Accessing elements

Individual elements in a list are accessed according to their position, called the index. The index of the first element is 0, the index of the second element is 1, and so forth. Negative indices refer to items at the end of the list. To get a particular element, write the name of the list and then the index of the element in square brackets.

Getting the first element

```
first_user = users[0]
```

Getting the second element

```
second_user = users[1]
```

Getting the last element

```
newest_user = users[-1]
```

Modifying individual items

Once you've defined a list, you can change individual elements in the list. You do this by referring to the index of the item you want to modify.

Changing an element

```
users[0] = 'valerie'
users[-2] = 'ronald'
```

Adding elements

You can add elements to the end of a list, or you can insert them wherever you like in a list.

Adding an element to the end of the list

```
users.append('amy')
```

Starting with an empty list

```
users = []
users.append('val')
users.append('bob')
users.append('mia')
```

Inserting elements at a particular position

```
users.insert(0, 'joe')
users.insert(3, 'bea')
```

Removing elements

You can remove elements by their position in a list, or by the value of the item. If you remove an item by its value, Python removes only the first item that has that value.

Deleting an element by its position

```
del users[-1]
```

Removing an item by its value

```
users.remove('mia')
```

Popping elements

If you want to work with an element that you're removing from the list, you can "pop" the element. If you think of the list as a stack of items, pop() takes an item off the top of the stack. By default pop() returns the last element in the list, but you can also pop elements from any position in the list.

Pop the last item from a list

```
most_recent_user = users.pop()
print(most_recent_user)
```

Pop the first item in a list

```
first_user = users.pop(0)
print(first_user)
```

List length

The len() function returns the number of items in a list.

Find the length of a list

```
num_users = len(users)
print("We have " + str(num_users) + " users.")
```

Sorting a list

The sort() method changes the order of a list permanently. The sorted() function returns a copy of the list, leaving the original list unchanged. You can sort the items in a list in alphabetical order, or reverse alphabetical order. You can also reverse the original order of the list. Keep in mind that lowercase and uppercase letters may affect the sort order.

Sorting a list permanently

```
users.sort()
```

Sorting a list permanently in reverse alphabetical order

```
users.sort(reverse=True)
```

Sorting a list temporarily

```
print(sorted(users))
print(sorted(users, reverse=True))
```

Reversing the order of a list

```
users.reverse()
```

Looping through a list

Lists can contain millions of items, so Python provides an efficient way to loop through all the items in a list. When you set up a loop, Python pulls each item from the list one at a time and stores it in a temporary variable, which you provide a name for. This name should be the singular version of the list name.

The indented block of code makes up the body of the loop, where you can work with each individual item. Any lines that are not indented run after the loop is completed.

Printing all items in a list

```
for user in users:
    print(user)
```

Printing a message for each item, and a separate message afterwards

```
for user in users:
    print("Welcome, " + user + "!")

print("Welcome, we're glad to see you all!")
```

The range() function

You can use the range() function to work with a set of numbers efficiently. The range() function starts at 0 by default, and stops one number below the number passed to it. You can use the list() function to efficiently generate a large list of numbers.

Printing the numbers 0 to 1000

```
for number in range(1001):
    print(number)
```

Printing the numbers 1 to 1000

```
for number in range(1, 1001):
    print(number)
```

Making a list of numbers from 1 to a million

```
numbers = list(range(1, 1000001))
```

Simple statistics

There are a number of simple statistics you can run on a list containing numerical data.

Finding the minimum value in a list

```
ages = [93, 99, 66, 17, 85, 1, 35, 82, 2, 77]
youngest = min(ages)
```

Finding the maximum value

```
ages = [93, 99, 66, 17, 85, 1, 35, 82, 2, 77]
oldest = max(ages)
```

Finding the sum of all values

```
ages = [93, 99, 66, 17, 85, 1, 35, 82, 2, 77]
total_years = sum(ages)
```

Slicing a list

You can work with any set of elements from a list. A portion of a list is called a slice. To slice a list start with the index of the first item you want, then add a colon and the index after the last item you want. Leave off the first index to start at the beginning of the list, and leave off the last index to slice through the end of the list.

Getting the first three items

```
finishers = ['kai', 'abe', 'ada', 'gus', 'zoe']
first_three = finishers[:3]
```

Getting the middle three items

```
middle_three = finishers[1:4]
```

Getting the last three items

```
last_three = finishers[-3:]
```

Copying a list

To copy a list make a slice that starts at the first item and ends at the last item. If you try to copy a list without using this approach, whatever you do to the copied list will affect the original list as well.

Making a copy of a list

```
finishers = ['kai', 'abe', 'ada', 'gus', 'zoe']
copy_of_finishers = finishers[:]
```

List comprehensions

You can use a loop to generate a list based on a range of numbers or on another list. This is a common operation, so Python offers a more efficient way to do it. List comprehensions may look complicated at first; if so, use the for loop approach until you're ready to start using comprehensions.

To write a comprehension, define an expression for the values you want to store in the list. Then write a for loop to generate input values needed to make the list.

Using a loop to generate a list of square numbers

```
squares = []
for x in range(1, 11):
    square = x**2
    squares.append(square)
```

Using a comprehension to generate a list of square numbers

```
squares = [x**2 for x in range(1, 11)]
```

Using a loop to convert a list of names to upper case

```
names = ['kai', 'abe', 'ada', 'gus', 'zoe']

upper_names = []
for name in names:
    upper_names.append(name.upper())
```

Using a comprehension to convert a list of names to upper case

```
names = ['kai', 'abe', 'ada', 'gus', 'zoe']

upper_names = [name.upper() for name in names]
```

Styling your code

Readability counts

- Use four spaces per indentation level.
- Keep your lines to 79 characters or fewer.
- Use single blank lines to group parts of your program visually.

Tuples

A tuple is like a list, except you can't change the values in a tuple once it's defined. Tuples are good for storing information that shouldn't be changed throughout the life of a program. Tuples are designated by parentheses instead of square brackets. (You can overwrite an entire tuple, but you can't change the individual elements in a tuple.)

Defining a tuple

```
dimensions = (800, 600)
```

Looping through a tuple

```
for dimension in dimensions:
    print(dimension)
```

Overwriting a tuple

```
dimensions = (800, 600)
print(dimensions)

dimensions = (1200, 900)
```

Visualizing your code

When you're first learning about data structures such as lists, it helps to visualize how Python is working with the information in your program. pythontutor.com is a great tool for seeing how Python keeps track of the information in a list. Try running the following code on pythontutor.com, and then run your own code.

Build a list and print the items in the list

```
dogs = []
dogs.append('willie')
dogs.append('hootz')
dogs.append('peso')
dogs.append('goblin')

for dog in dogs:
    print("Hello " + dog + "!")
print("I love these dogs!")

print("\nThese were my first two dogs:")
old_dogs = dogs[:2]
for old_dog in old_dogs:
    print(old_dog)

del dogs[0]
dogs.remove('peso')
print(dogs)
```

More cheat sheets available at
ehmatthes.github.io/pcc/

Beginner's Python Cheat Sheet — Dictionaries

What are dictionaries?

Python's dictionaries allow you to connect pieces of related information. Each piece of information in a dictionary is stored as a key-value pair. When you provide a key, Python returns the value associated with that key. You can loop through all the key-value pairs, all the keys, or all the values.

Defining a dictionary

Use curly braces to define a dictionary. Use colons to connect keys and values, and use commas to separate individual key-value pairs.

Making a dictionary

```
alien_0 = {'color': 'green', 'points': 5}
```

Accessing values

To access the value associated with an individual key give the name of the dictionary and then place the key in a set of square brackets. If the key you're asking for is not in the dictionary, an error will occur.

You can also use the get() method, which returns None instead of an error if the key doesn't exist. You can also specify a default value to use if the key is not in the dictionary.

Getting the value associated with a key

```
alien_0 = {'color': 'green', 'points': 5}

print(alien_0['color'])
print(alien_0['points'])
```

Getting the value with get()

```
alien_0 = {'color': 'green'}

alien_color = alien_0.get('color')
alien_points = alien_0.get('points', 0)

print(alien_color)
print(alien_points)
```

Adding new key-value pairs

You can store as many key-value pairs as you want in a dictionary, until your computer runs out of memory. To add a new key-value pair to an existing dictionary give the name of the dictionary and the new key in square brackets, and set it equal to the new value.

This also allows you to start with an empty dictionary and add key-value pairs as they become relevant.

Adding a key-value pair

```
alien_0 = {'color': 'green', 'points': 5}

alien_0['x'] = 0
alien_0['y'] = 25
alien_0['speed'] = 1.5
```

Adding to an empty dictionary

```
alien_0 = {}
alien_0['color'] = 'green'
alien_0['points'] = 5
```

Modifying values

You can modify the value associated with any key in a dictionary. To do so give the name of the dictionary and enclose the key in square brackets, then provide the new value for that key.

Modifying values in a dictionary

```
alien_0 = {'color': 'green', 'points': 5}
print(alien_0)

# Change the alien's color and point value.
alien_0['color'] = 'yellow'
alien_0['points'] = 10
print(alien_0)
```

Removing key-value pairs

You can remove any key-value pair you want from a dictionary. To do so use the del keyword and the dictionary name, followed by the key in square brackets. This will delete the key and its associated value.

Deleting a key-value pair

```
alien_0 = {'color': 'green', 'points': 5}
print(alien_0)

del alien_0['points']
print(alien_0)
```

Visualizing dictionaries

Try running some of these examples on pythontutor.com.

Looping through a dictionary

You can loop through a dictionary in three ways: you can loop through all the key-value pairs, all the keys, or all the values.

A dictionary only tracks the connections between keys and values; it doesn't track the order of items in the dictionary. If you want to process the information in order, you can sort the keys in your loop.

Looping through all key-value pairs

```
# Store people's favorite languages.
fav_languages = {
    'jen': 'python',
    'sarah': 'c',
    'edward': 'ruby',
    'phil': 'python',
    }

# Show each person's favorite language.
for name, language in fav_languages.items():
    print(name + ": " + language)
```

Looping through all the keys

```
# Show everyone who's taken the survey.
for name in fav_languages.keys():
    print(name)
```

Looping through all the values

```
# Show all the languages that have been chosen.
for language in fav_languages.values():
    print(language)
```

Looping through all the keys in order

```
# Show each person's favorite language,
#   in order by the person's name.
for name in sorted(fav_languages.keys()):
    print(name + ": " + language)
```

Dictionary length

You can find the number of key-value pairs in a dictionary.

Finding a dictionary's length

```
num_responses = len(fav_languages)
```

Nesting — A list of dictionaries

It's sometimes useful to store a set of dictionaries in a list; this is called nesting.

Storing dictionaries in a list

```
# Start with an empty list.
users = []

# Make a new user, and add them to the list.
new_user = {
    'last': 'fermi',
    'first': 'enrico',
    'username': 'efermi',
    }
users.append(new_user)

# Make another new user, and add them as well.
new_user = {
    'last': 'curie',
    'first': 'marie',
    'username': 'mcurie',
    }
users.append(new_user)

# Show all information about each user.
for user_dict in users:
    for k, v in user_dict.items():
        print(k + ": " + v)
    print("\n")
```

You can also define a list of dictionaries directly, without using append():

```
# Define a list of users, where each user
#   is represented by a dictionary.
users = [
    {
        'last': 'fermi',
        'first': 'enrico',
        'username': 'efermi',
    },
    {
        'last': 'curie',
        'first': 'marie',
        'username': 'mcurie',
    },
]

# Show all information about each user.
for user_dict in users:
    for k, v in user_dict.items():
        print(k + ": " + v)
    print("\n")
```

Nesting — Lists in a dictionary

Storing a list inside a dictionary alows you to associate more than one value with each key.

Storing lists in a dictionary

```
# Store multiple languages for each person.
fav_languages = {
    'jen': ['python', 'ruby'],
    'sarah': ['c'],
    'edward': ['ruby', 'go'],
    'phil': ['python', 'haskell'],
}

# Show all responses for each person.
for name, langs in fav_languages.items():
    print(name + ": ")
    for lang in langs:
        print("- " + lang)
```

Nesting — A dictionary of dictionaries

You can store a dictionary inside another dictionary. In this case each value associated with a key is itself a dictionary.

Storing dictionaries in a dictionary

```
users = {
    'aeinstein': {
        'first': 'albert',
        'last': 'einstein',
        'location': 'princeton',
        },
    'mcurie': {
        'first': 'marie',
        'last': 'curie',
        'location': 'paris',
        },
    }

for username, user_dict in users.items():
    print("\nUsername: " + username)
    full_name = user_dict['first'] + " "
    full_name += user_dict['last']
    location = user_dict['location']

    print("\tFull name: " + full_name.title())
    print("\tLocation: " + location.title())
```

Levels of nesting

Nesting is extremely useful in certain situations. However, be aware of making your code overly complex. If you're nesting items much deeper than what you see here there are probably simpler ways of managing your data, such as using classes.

Using an OrderedDict

Standard Python dictionaries don't keep track of the order in which keys and values are added; they only preserve the association between each key and its value. If you want to preserve the order in which keys and values are added, use an OrderedDict.

Preserving the order of keys and values

```
from collections import OrderedDict

# Store each person's languages, keeping
#  track of who respoded first.
fav_languages = OrderedDict()

fav_languages['jen'] = ['python', 'ruby']
fav_languages['sarah'] = ['c']
fav_languages['edward'] = ['ruby', 'go']
fav_languages['phil'] = ['python', 'haskell']

# Display the results, in the same order they
#  were entered.
for name, langs in fav_languages.items():
    print(name + ":")
    for lang in langs:
        print("- " + lang)
```

Generating a million dictionaries

You can use a loop to generate a large number of dictionaries efficiently, if all the dictionaries start out with similar data.

A million aliens

```
aliens = []

# Make a million green aliens, worth 5 points
#  each. Have them all start in one row.
for alien_num in range(1000000):
    new_alien = {}
    new_alien['color'] = 'green'
    new_alien['points'] = 5
    new_alien['x'] = 20 * alien_num
    new_alien['y'] = 0
    aliens.append(new_alien)

# Prove the list contains a million aliens.
num_aliens = len(aliens)

print("Number of aliens created:")
print(num_aliens)
```

Beginner's Python Cheat Sheet — If Statements and While Loops

What are if statements? What are while loops?

If statements allow you to examine the current state of a program and respond appropriately to that state. You can write a simple if statement that checks one condition, or you can create a complex series of if statements that idenitfy the exact conditions you're looking for.

While loops run as long as certain conditions remain true. You can use while loops to let your programs run as long as your users want them to.

Conditional Tests

A conditional test is an expression that can be evaluated as True or False. Python uses the values True and False to decide whether the code in an if statement should be executed.

Checking for equality

A single equal sign assigns a value to a variable. A double equal sign (==) checks whether two values are equal.

```
>>> car = 'bmw'
>>> car == 'bmw'
True
>>> car = 'audi'
>>> car == 'bmw'
False
```

Ignoring case when making a comparison

```
>>> car = 'Audi'
>>> car.lower() == 'audi'
True
```

Checking for inequality

```
>>> topping = 'mushrooms'
>>> topping != 'anchovies'
True
```

Numerical comparisons

Testing numerical values is similar to testing string values.

Testing equality and inequality

```
>>> age = 18
>>> age == 18
True
>>> age != 18
False
```

Comparison operators

```
>>> age = 19
>>> age < 21
True
>>> age <= 21
True
>>> age > 21
False
>>> age >= 21
False
```

Checking multiple conditions

You can check multiple conditions at the same time. The and operator returns True if all the conditions listed are True. The or operator returns True if any condition is True.

Using and to check multiple conditions

```
>>> age_0 = 22
>>> age_1 = 18
>>> age_0 >= 21 and age_1 >= 21
False
>>> age_1 = 23
>>> age_0 >= 21 and age_1 >= 21
True
```

Using or to check multiple conditions

```
>>> age_0 = 22
>>> age_1 = 18
>>> age_0 >= 21 or age_1 >= 21
True
>>> age_0 = 18
>>> age_0 >= 21 or age_1 >= 21
False
```

Boolean values

A boolean value is either True or False. Variables with boolean values are often used to keep track of certain conditions within a program.

Simple boolean values

```
game_active = True
can_edit = False
```

If statements

Several kinds of if statements exist. Your choice of which to use depends on the number of conditions you need to test. You can have as many elif blocks as you need, and the else block is always optional.

Simple if statement

```
age = 19

if age >= 18:
    print("You're old enough to vote!")
```

If-else statements

```
age = 17

if age >= 18:
    print("You're old enough to vote!")
else:
    print("You can't vote yet.")
```

The if-elif-else chain

```
age = 12

if age < 4:
   price = 0
elif age < 18:
    price = 5
else:
    price = 10

print("Your cost is $" + str(price) + ".")
```

Conditional tests with lists

You can easily test whether a certain value is in a list. You can also test whether a list is empty before trying to loop through the list.

Testing if a value is in a list

```
>>> players = ['al', 'bea', 'cyn', 'dale']
>>> 'al' in players
True
>>> 'eric' in players
False
```

Conditional tests with lists (cont.)

Testing if a value is not in a list

```
banned_users = ['ann', 'chad', 'dee']
user = 'erin'

if user not in banned_users:
    print("You can play!")
```

Checking if a list is empty

```
players = []

if players:
    for player in players:
        print("Player: " + player.title())
else:
    print("We have no players yet!")
```

Accepting input

You can allow your users to enter input using the input() statement. In Python 3, all input is stored as a string.

Simple input

```
name = input("What's your name? ")
print("Hello, " + name + ".")
```

Accepting numerical input

```
age = input("How old are you? ")
age = int(age)

if age >= 18:
    print("\nYou can vote!")
else:
    print("\nYou can't vote yet.")
```

Accepting input in Python 2.7

Use raw_input() in Python 2.7. This function interprets all input as a string, just as input() does in Python 3.

```
name = raw_input("What's your name? ")
print("Hello, " + name + ".")
```

While loops

A while loop repeats a block of code as long as a condition is True.

Counting to 5

```
current_number = 1

while current_number <= 5:
    print(current_number)
    current_number += 1
```

While loops (cont.)

Letting the user choose when to quit

```
prompt = "\nTell me something, and I'll "
prompt += "repeat it back to you."
prompt += "\nEnter 'quit' to end the program. "

message = ""
while message != 'quit':
    message = input(prompt)

    if message != 'quit':
        print(message)
```

Using a flag

```
prompt = "\nTell me something, and I'll "
prompt += "repeat it back to you."
prompt += "\nEnter 'quit' to end the program. "

active = True
while active:
    message = input(prompt)

    if message == 'quit':
        active = False
    else:
        print(message)
```

Using break to exit a loop

```
prompt = "\nWhat cities have you visited?"
prompt += "\nEnter 'quit' when you're done. "

while True:
    city = input(prompt)

    if city == 'quit':
        break
    else:
        print("I've been to " + city + "!")
```

Accepting input with Sublime Text

Sublime Text doesn't run programs that prompt the user for input. You can use Sublime Text to write programs that prompt for input, but you'll need to run these programs from a terminal.

Breaking out of loops

You can use the break statement and the continue statement with any of Python's loops. For example you can use break to quit a for loop that's working through a list or a dictionary. You can use continue to skip over certain items when looping through a list or dictionary as well.

While loops (cont.)

Using continue in a loop

```
banned_users = ['eve', 'fred', 'gary', 'helen']

prompt = "\nAdd a player to your team."
prompt += "\nEnter 'quit' when you're done. "

players = []
while True:
    player = input(prompt)
    if player == 'quit':
        break
    elif player in banned_users:
        print(player + " is banned!")
        continue
    else:
        players.append(player)

print("\nYour team:")
for player in players:
    print(player)
```

Avoiding infinite loops

Every while loop needs a way to stop running so it won't continue to run forever. If there's no way for the condition to become False, the loop will never stop running.

An infinite loop

```
while True:
    name = input("\nWho are you? ")
    print("Nice to meet you, " + name + "!")
```

Removing all instances of a value from a list

The remove() method removes a specific value from a list, but it only removes the first instance of the value you provide. You can use a while loop to remove all instances of a particular value.

Removing all cats from a list of pets

```
pets = ['dog', 'cat', 'dog', 'fish', 'cat',
        'rabbit', 'cat']
print(pets)

while 'cat' in pets:
    pets.remove('cat')

print(pets)
```

More cheat sheets available at
ehmatthes.github.io/pcc/

Beginner's Python Cheat Sheet — Functions

What are functions?

Functions are named blocks of code designed to do one specific job. Functions allow you to write code once that can then be run whenever you need to accomplish the same task. Functions can take in the information they need, and return the information they generate. Using functions effectively makes your programs easier to write, read, test, and fix.

Defining a function

The first line of a function is its definition, marked by the keyword def. The name of the function is followed by a set of parentheses and a colon. A docstring, in triple quotes, describes what the function does. The body of a function is indented one level.

To call a function, give the name of the function followed by a set of parentheses.

Making a function

```
def greet_user():
    """Display a simple greeting."""
    print("Hello!")

greet_user()
```

Passing information to a function

Information that's passed to a function is called an argument; information that's received by a function is called a parameter. Arguments are included in parentheses after the function's name, and parameters are listed in parentheses in the function's definition.

Passing a single argument

```
def greet_user(username):
    """Display a simple greeting."""
    print("Hello, " + username + "!")

greet_user('jesse')
greet_user('diana')
greet_user('brandon')
```

Positional and keyword arguments

The two main kinds of arguments are positional and keyword arguments. When you use positional arguments Python matches the first argument in the function call with the first parameter in the function definition, and so forth.

With keyword arguments, you specify which parameter each argument should be assigned to in the function call. When you use keyword arguments, the order of the arguments doesn't matter.

Using positional arguments

```
def describe_pet(animal, name):
    """Display information about a pet."""
    print("\nI have a " + animal + ".")
    print("Its name is " + name + ".")

describe_pet('hamster', 'harry')
describe_pet('dog', 'willie')
```

Using keyword arguments

```
def describe_pet(animal, name):
    """Display information about a pet."""
    print("\nI have a " + animal + ".")
    print("Its name is " + name + ".")

describe_pet(animal='hamster', name='harry')
describe_pet(name='willie', animal='dog')
```

Default values

You can provide a default value for a parameter. When function calls omit this argument the default value will be used. Parameters with default values must be listed after parameters without default values in the function's definition so positional arguments can still work correctly.

Using a default value

```
def describe_pet(name, animal='dog'):
    """Display information about a pet."""
    print("\nI have a " + animal + ".")
    print("Its name is " + name + ".")

describe_pet('harry', 'hamster')
describe_pet('willie')
```

Using None to make an argument optional

```
def describe_pet(animal, name=None):
    """Display information about a pet."""
    print("\nI have a " + animal + ".")
    if name:
        print("Its name is " + name + ".")

describe_pet('hamster', 'harry')
describe_pet('snake')
```

Return values

A function can return a value or a set of values. When a function returns a value, the calling line must provide a variable in which to store the return value. A function stops running when it reaches a return statement.

Returning a single value

```
def get_full_name(first, last):
    """Return a neatly formatted full name."""
    full_name = first + ' ' + last
    return full_name.title()

musician = get_full_name('jimi', 'hendrix')
print(musician)
```

Returning a dictionary

```
def build_person(first, last):
    """Return a dictionary of information
    about a person.
    """
    person = {'first': first, 'last': last}
    return person

musician = build_person('jimi', 'hendrix')
print(musician)
```

Returning a dictionary with optional values

```
def build_person(first, last, age=None):
    """Return a dictionary of information
    about a person.
    """
    person = {'first': first, 'last': last}
    if age:
        person['age'] = age
    return person

musician = build_person('jimi', 'hendrix', 27)
print(musician)

musician = build_person('janis', 'joplin')
print(musician)
```

Visualizing functions

Try running some of these examples on pythontutor.com.

Passing a list to a function

You can pass a list as an argument to a function, and the function can work with the values in the list. Any changes the function makes to the list will affect the original list. You can prevent a function from modifying a list by passing a copy of the list as an argument.

Passing a list as an argument

```
def greet_users(names):
    """Print a simple greeting to everyone."""
    for name in names:
        msg = "Hello, " + name + "!"
        print(msg)

usernames = ['hannah', 'ty', 'margot']
greet_users(usernames)
```

Allowing a function to modify a list

The following example sends a list of models to a function for printing. The original list is emptied, and the second list is filled.

```
def print_models(unprinted, printed):
    """3d print a set of models."""
    while unprinted:
        current_model = unprinted.pop()
        print("Printing " + current_model)
        printed.append(current_model)

# Store some unprinted designs,
#  and print each of them.
unprinted = ['phone case', 'pendant', 'ring']
printed = []
print_models(unprinted, printed)

print("\nUnprinted:", unprinted)
print("Printed:", printed)
```

Preventing a function from modifying a list

The following example is the same as the previous one, except the original list is unchanged after calling print_models().

```
def print_models(unprinted, printed):
    """3d print a set of models."""
    while unprinted:
        current_model = unprinted.pop()
        print("Printing " + current_model)
        printed.append(current_model)

# Store some unprinted designs,
#  and print each of them.
original = ['phone case', 'pendant', 'ring']
printed = []

print_models(original[:], printed)
print("\nOriginal:", original)
print("Printed:", printed)
```

Passing an arbitrary number of arguments

*Sometimes you won't know how many arguments a function will need to accept. Python allows you to collect an arbitrary number of arguments into one parameter using the * operator. A parameter that accepts an arbitrary number of arguments must come last in the function definition.*

*The ** operator allows a parameter to collect an arbitrary number of keyword arguments.*

Collecting an arbitrary number of arguments

```
def make_pizza(size, *toppings):
    """Make a pizza."""
    print("\nMaking a " + size + " pizza.")
    print("Toppings:")
    for topping in toppings:
        print("- " + topping)

# Make three pizzas with different toppings.
make_pizza('small', 'pepperoni')
make_pizza('large', 'bacon bits', 'pineapple')
make_pizza('medium', 'mushrooms', 'peppers',
        'onions', 'extra cheese')
```

Collecting an arbitrary number of keyword arguments

```
def build_profile(first, last, **user_info):
    """Build a user's profile dictionary."""
    # Build a dict with the required keys.
    profile = {'first': first, 'last': last}

    # Add any other keys and values.
    for key, value in user_info.items():
        profile[key] = value

    return profile

# Create two users with different kinds
#   of information.
user_0 = build_profile('albert', 'einstein',
        location='princeton')
user_1 = build_profile('marie', 'curie',
        location='paris', field='chemistry')

print(user_0)
print(user_1)
```

What's the best way to structure a function?

As you can see there are many ways to write and call a function. When you're starting out, aim for something that simply works. As you gain experience you'll develop an understanding of the more subtle advantages of different structures such as positional and keyword arguments, and the various approaches to importing functions. For now if your functions do what you need them to, you're doing well.

Modules

You can store your functions in a separate file called a module, and then import the functions you need into the file containing your main program. This allows for cleaner program files. (Make sure your module is stored in the same directory as your main program.)

Storing a function in a module

File: pizza.py

```
def make_pizza(size, *toppings):
    """Make a pizza."""
    print("\nMaking a " + size + " pizza.")
    print("Toppings:")
    for topping in toppings:
        print("- " + topping)
```

Importing an entire module

File: making_pizzas.py
Every function in the module is available in the program file.

```
import pizza

pizza.make_pizza('medium', 'pepperoni')
pizza.make_pizza('small', 'bacon', 'pineapple')
```

Importing a specific function

Only the imported functions are available in the program file.

```
from pizza import make_pizza

make_pizza('medium', 'pepperoni')
make_pizza('small', 'bacon', 'pineapple')
```

Giving a module an alias

```
import pizza as p

p.make_pizza('medium', 'pepperoni')
p.make_pizza('small', 'bacon', 'pineapple')
```

Giving a function an alias

```
from pizza import make_pizza as mp

mp('medium', 'pepperoni')
mp('small', 'bacon', 'pineapple')
```

Importing all functions from a module

Don't do this, but recognize it when you see it in others' code. It can result in naming conflicts, which can cause errors.

```
from pizza import *

make_pizza('medium', 'pepperoni')
make_pizza('small', 'bacon', 'pineapple')
```

More cheat sheets available at
ehmatthes.github.io/pcc/

Beginner's Python Cheat Sheet - Classes

What are classes?

Classes are the foundation of object-oriented programming. Classes represent real-world things you want to model in your programs: for example dogs, cars, and robots. You use a class to make objects, which are specific instances of dogs, cars, and robots. A class defines the general behavior that a whole category of objects can have, and the information that can be associated with those objects.

Classes can inherit from each other – you can write a class that extends the functionality of an existing class. This allows you to code efficiently for a wide variety of situations.

Creating and using a class

Consider how we might model a car. What information would we associate with a car, and what behavior would it have? The information is stored in variables called attributes, and the behavior is represented by functions. Functions that are part of a class are called methods.

The Car class

```
class Car():
    """A simple attempt to model a car."""

    def __init__(self, make, model, year):
        """Initialize car attributes."""
        self.make = make
        self.model = model
        self.year = year

        # Fuel capacity and level in gallons.
        self.fuel_capacity = 15
        self.fuel_level = 0

    def fill_tank(self):
        """Fill gas tank to capacity."""
        self.fuel_level = self.fuel_capacity
        print("Fuel tank is full.")

    def drive(self):
        """Simulate driving."""
        print("The car is moving.")
```

Creating and using a class (cont.)

Creating an object from a class

```
my_car = Car('audi', 'a4', 2016)
```

Accessing attribute values

```
print(my_car.make)
print(my_car.model)
print(my_car.year)
```

Calling methods

```
my_car.fill_tank()
my_car.drive()
```

Creating multiple objects

```
my_car = Car('audi', 'a4', 2016)
my_old_car = Car('subaru', 'outback', 2013)
my_truck = Car('toyota', 'tacoma', 2010)
```

Modifying attributes

You can modify an attribute's value directly, or you can write methods that manage updating values more carefully.

Modifying an attribute directly

```
my_new_car = Car('audi', 'a4', 2016)
my_new_car.fuel_level = 5
```

Writing a method to update an attribute's value

```
def update_fuel_level(self, new_level):
    """Update the fuel level."""
    if new_level <= self.fuel_capacity:
        self.fuel_level = new_level
    else:
        print("The tank can't hold that much!")
```

Writing a method to increment an attribute's value

```
def add_fuel(self, amount):
    """Add fuel to the tank."""
    if (self.fuel_level + amount
            <= self.fuel_capacity):
        self.fuel_level += amount
        print("Added fuel.")
    else:
        print("The tank won't hold that much.")
```

Naming conventions

In Python class names are written in CamelCase and object names are written in lowercase with underscores. Modules that contain classes should still be named in lowercase with underscores.

Class inheritance

If the class you're writing is a specialized version of another class, you can use inheritance. When one class inherits from another, it automatically takes on all the attributes and methods of the parent class. The child class is free to introduce new attributes and methods, and override attributes and methods of the parent class.

To inherit from another class include the name of the parent class in parentheses when defining the new class.

The __init__() method for a child class

```
class ElectricCar(Car):
    """A simple model of an electric car."""

    def __init__(self, make, model, year):
        """Initialize an electric car."""
        super().__init__(make, model, year)

        # Attributes specific to electric cars.
        # Battery capacity in kWh.
        self.battery_size = 70
        # Charge level in %.
        self.charge_level = 0
```

Adding new methods to the child class

```
class ElectricCar(Car):
    --snip--
    def charge(self):
        """Fully charge the vehicle."""
        self.charge_level = 100
        print("The vehicle is fully charged.")
```

Using child methods and parent methods

```
my_ecar = ElectricCar('tesla', 'model s', 2016)

my_ecar.charge()
my_ecar.drive()
```

Finding your workflow

There are many ways to model real world objects and situations in code, and sometimes that variety can feel overwhelming. Pick an approach and try it – if your first attempt doesn't work, try a different approach.

Class inheritance (cont.)

Overriding parent methods

```
class ElectricCar(Car):
    --snip--
    def fill_tank(self):
        """Display an error message."""
        print("This car has no fuel tank!")
```

Instances as attributes

A class can have objects as attributes. This allows classes to work together to model complex situations.

A Battery class

```
class Battery():
    """A battery for an electric car."""

    def __init__(self, size=70):
        """Initialize battery attributes."""
        # Capacity in kWh, charge level in %.
        self.size = size
        self.charge_level = 0

    def get_range(self):
        """Return the battery's range."""
        if self.size == 70:
            return 240
        elif self.size == 85:
            return 270
```

Using an instance as an attribute

```
class ElectricCar(Car):
    --snip--

    def __init__(self, make, model, year):
        """Initialize an electric car."""
        super().__init__(make, model, year)

        # Attribute specific to electric cars.
        self.battery = Battery()

    def charge(self):
        """Fully charge the vehicle."""
        self.battery.charge_level = 100
        print("The vehicle is fully charged.")
```

Using the instance

```
my_ecar = ElectricCar('tesla', 'model x', 2016)

my_ecar.charge()
print(my_ecar.battery.get_range())
my_ecar.drive()
```

Importing classes

Class files can get long as you add detailed information and functionality. To help keep your program files uncluttered, you can store your classes in modules and import the classes you need into your main program.

Storing classes in a file

car.py

```
"""Represent gas and electric cars."""

class Car():
    """A simple attempt to model a car."""
    --snip—

class Battery():
    """A battery for an electric car."""
    --snip--

class ElectricCar(Car):
    """A simple model of an electric car."""
    --snip--
```

Importing individual classes from a module

my_cars.py

```
from car import Car, ElectricCar

my_beetle = Car('volkswagen', 'beetle', 2016)
my_beetle.fill_tank()
my_beetle.drive()

my_tesla = ElectricCar('tesla', 'model s',
2016)
my_tesla.charge()
my_tesla.drive()
```

Importing an entire module

```
import car

my_beetle = car.Car(
        'volkswagen', 'beetle', 2016)
my_beetle.fill_tank()
my_beetle.drive()

my_tesla = car.ElectricCar(
        'tesla', 'model s', 2016)
my_tesla.charge()
my_tesla.drive()
```

Importing all classes from a module

(Don't do this, but recognize it when you see it.)

```
from car import *

my_beetle = Car('volkswagen', 'beetle', 2016)
```

Classes in Python 2.7

Classes should inherit from object

```
class ClassName(object):
```

The Car class in Python 2.7

```
class Car(object):
```

Child class __init__() method is different

```
class ChildClassName(ParentClass):
    def __init__(self):
        super(ClassName, self).__init__()
```

The ElectricCar class in Python 2.7

```
class ElectricCar(Car):
    def __init__(self, make, model, year):
        super(ElectricCar, self).__init__(
                make, model, year)
```

Storing objects in a list

A list can hold as many items as you want, so you can make a large number of objects from a class and store them in a list.

Here's an example showing how to make a fleet of rental cars, and make sure all the cars are ready to drive.

A fleet of rental cars

```
from car import Car, ElectricCar

# Make lists to hold a fleet of cars.
gas_fleet = []
electric_fleet = []

# Make 500 gas cars and 250 electric cars.
for _ in range(500):
    car = Car('ford', 'focus', 2016)
    gas_fleet.append(car)
for _ in range(250):
    ecar = ElectricCar('nissan', 'leaf', 2016)
    electric_fleet.append(ecar)

# Fill the gas cars, and charge electric cars.
for car in gas_fleet:
    car.fill_tank()
for ecar in electric_fleet:
    ecar.charge()

print("Gas cars:", len(gas_fleet))
print("Electric cars:", len(electric_fleet))
```

More cheat sheets available at
ehmatthes.github.io/pcc/

Beginner's Python Cheat Sheet — Files and Exceptions

What are files? What are exceptions?

Your programs can read information in from files, and they can write data to files. Reading from files allows you to work with a wide variety of information; writing to files allows users to pick up where they left off the next time they run your program. You can write text to files, and you can store Python structures such as lists in data files.

Exceptions are special objects that help your programs respond to errors in appropriate ways. For example if your program tries to open a file that doesn't exist, you can use exceptions to display an informative error message instead of having the program crash.

Reading from a file

To read from a file your program needs to open the file and then read the contents of the file. You can read the entire contents of the file at once, or read the file line by line. The with statement makes sure the file is closed properly when the program has finished accessing the file.

Reading an entire file at once

```
filename = 'siddhartha.txt'

with open(filename) as f_obj:
    contents = f_obj.read()

print(contents)
```

Reading line by line

Each line that's read from the file has a newline character at the end of the line, and the print function adds its own newline character. The rstrip() method gets rid of the the extra blank lines this would result in when printing to the terminal.

```
filename = 'siddhartha.txt'

with open(filename) as f_obj:
    for line in f_obj:
        print(line.rstrip())
```

Reading from a file (cont.)

Storing the lines in a list

```
filename = 'siddhartha.txt'

with open(filename) as f_obj:
    lines = f_obj.readlines()

for line in lines:
    print(line.rstrip())
```

Writing to a file

Passing the 'w' argument to open() tells Python you want to write to the file. Be careful; this will erase the contents of the file if it already exists. Passing the 'a' argument tells Python you want to append to the end of an existing file.

Writing to an empty file

```
filename = 'programming.txt'

with open(filename, 'w') as f:
    f.write("I love programming!")
```

Writing multiple lines to an empty file

```
filename = 'programming.txt'

with open(filename, 'w') as f:
    f.write("I love programming!\n")
    f.write("I love creating new games.\n")
```

Appending to a file

```
filename = 'programming.txt'

with open(filename, 'a') as f:
    f.write("I also love working with data.\n")
    f.write("I love making apps as well.\n")
```

File paths

When Python runs the open() function, it looks for the file in the same directory where the program that's being excuted is stored. You can open a file from a subfolder using a relative path. You can also use an absolute path to open any file on your system.

Opening a file from a subfolder

```
f_path = "text_files/alice.txt"

with open(f_path) as f_obj:
    lines = f_obj.readlines()

for line in lines:
    print(line.rstrip())
```

File paths (cont.)

Opening a file using an absolute path

```
f_path = "/home/ehmatthes/books/alice.txt"

with open(f_path) as f_obj:
    lines = f_obj.readlines()
```

Opening a file on Windows

Windows will sometimes interpret forward slashes incorrectly. If you run into this, use backslashes in your file paths.

```
f_path = "C:\Users\ehmatthes\books\alice.txt"

with open(f_path) as f_obj:
    lines = f_obj.readlines()
```

The try-except block

When you think an error may occur, you can write a try-except block to handle the exception that might be raised. The try block tells Python to try running some code, and the except block tells Python what to do if the code results in a particular kind of error.

Handling the ZeroDivisionError exception

```
try:
    print(5/0)
except ZeroDivisionError:
    print("You can't divide by zero!")
```

Handling the FileNotFoundError exception

```
f_name = 'siddhartha.txt'

try:
    with open(f_name) as f_obj:
        lines = f_obj.readlines()
except FileNotFoundError:
    msg = "Can't find file {0}.".format(f_name)
    print(msg)
```

Knowing which exception to handle

It can be hard to know what kind of exception to handle when writing code. Try writing your code without a try block, and make it generate an error. The traceback will tell you what kind of exception your program needs to handle.

The else block

The try block should only contain code that may cause an error. Any code that depends on the try block running successfully should be placed in the else block.

Using an else block

```
print("Enter two numbers. I'll divide them.")

x = input("First number: ")
y = input("Second number: ")

try:
    result = int(x) / int(y)
except ZeroDivisionError:
    print("You can't divide by zero!")
else:
    print(result)
```

Preventing crashes from user input

Without the except block in the following example, the program would crash if the user tries to divide by zero. As written, it will handle the error gracefully and keep running.

```
"""A simple calculator for division only."""

print("Enter two numbers. I'll divide them.")
print("Enter 'q' to quit.")

while True:
    x = input("\nFirst number: ")
    if x == 'q':
        break
    y = input("Second number: ")
    if y == 'q':
        break

    try:
        result = int(x) / int(y)
    except ZeroDivisionError:
        print("You can't divide by zero!")
    else:
        print(result)
```

Deciding which errors to report

Well-written, properly tested code is not very prone to internal errors such as syntax or logical errors. But every time your program depends on something external such as user input or the existence of a file, there's a possibility of an exception being raised.

It's up to you how to communicate errors to your users. Sometimes users need to know if a file is missing; sometimes it's better to handle the error silently. A little experience will help you know how much to report.

Failing silently

Sometimes you want your program to just continue running when it encounters an error, without reporting the error to the user. Using the pass statement in an else block allows you to do this.

Using the pass statement in an else block

```
f_names = ['alice.txt', 'siddhartha.txt',
        'moby_dick.txt', 'little_women.txt']

for f_name in f_names:
    # Report the length of each file found.
    try:
        with open(f_name) as f_obj:
            lines = f_obj.readlines()
    except FileNotFoundError:
        # Just move on to the next file.
        pass
    else:
        num_lines = len(lines)
        msg = "{0} has {1} lines.".format(
            f_name, num_lines)
        print(msg)
```

Avoid bare except blocks

Exception-handling code should catch specific exceptions that you expect to happen during your program's execution. A bare except block will catch all exceptions, including keyboard interrupts and system exits you might need when forcing a program to close.

If you want to use a try block and you're not sure which exception to catch, use Exception. It will catch most exceptions, but still allow you to interrupt programs intentionally.

Don't use bare except blocks

```
try:
    # Do something
except:
    pass
```

Use Exception instead

```
try:
    # Do something
except Exception:
    pass
```

Printing the exception

```
try:
    # Do something
except Exception as e:
    print(e, type(e))
```

Storing data with json

The json module allows you to dump simple Python data structures into a file, and load the data from that file the next time the program runs. The JSON data format is not specific to Python, so you can share this kind of data with people who work in other languages as well.

Knowing how to manage exceptions is important when working with stored data. You'll usually want to make sure the data you're trying to load exists before working with it.

Using json.dump() to store data

```
"""Store some numbers."""

import json

numbers = [2, 3, 5, 7, 11, 13]

filename = 'numbers.json'
with open(filename, 'w') as f_obj:
    json.dump(numbers, f_obj)
```

Using json.load() to read data

```
"""Load some previously stored numbers."""

import json

filename = 'numbers.json'
with open(filename) as f_obj:
    numbers = json.load(f_obj)

print(numbers)
```

Making sure the stored data exists

```
import json

f_name = 'numbers.json'

try:
    with open(f_name) as f_obj:
        numbers = json.load(f_obj)
except FileNotFoundError:
    msg = "Can't find {0}.".format(f_name)
    print(msg)
else:
    print(numbers)
```

Practice with exceptions

Take a program you've already written that prompts for user input, and add some error-handling code to the program.

More cheat sheets available at
ehmatthes.github.io/pcc/

Beginner's Python Cheat Sheet — Testing Your Code

Why test your code?

When you write a function or a class, you can also write tests for that code. Testing proves that your code works as it's supposed to in the situations it's designed to handle, and also when people use your programs in unexpected ways. Writing tests gives you confidence that your code will work correctly as more people begin to use your programs. You can also add new features to your programs and know that you haven't broken existing behavior.

A unit test verifies that one specific aspect of your code works as it's supposed to. A test case is a collection of unit tests which verify your code's behavior in a wide variety of situations.

Testing a function: A passing test

Python's unittest module provides tools for testing your code. To try it out, we'll create a function that returns a full name. We'll use the function in a regular program, and then build a test case for the function.

A function to test

Save this as full_names.py

```
def get_full_name(first, last):
    """Return a full name."""
    full_name = "{0} {1}".format(first, last)
    return full_name.title()
```

Using the function

Save this as names.py

```
from full_names import get_full_name

janis = get_full_name('janis', 'joplin')
print(janis)

bob = get_full_name('bob', 'dylan')
print(bob)
```

Testing a function (cont.)

Building a testcase with one unit test

To build a test case, make a class that inherits from `unittest.TestCase` *and write methods that begin with* `test_`. *Save this as test_full_names.py*

```
import unittest
from full_names import get_full_name

class NamesTestCase(unittest.TestCase):
    """Tests for names.py."""

    def test_first_last(self):
        """Test names like Janis Joplin."""
        full_name = get_full_name('janis',
                'joplin')
        self.assertEqual(full_name,
                'Janis Joplin')

unittest.main()
```

Running the test

Python reports on each unit test in the test case. The dot reports a single passing test. Python informs us that it ran 1 test in less than 0.001 seconds, and the OK lets us know that all unit tests in the test case passed.

```
.
--------------------------------------
Ran 1 test in 0.000s

OK
```

Testing a function: A failing test

Failing tests are important; they tell you that a change in the code has affected existing behavior. When a test fails, you need to modify the code so the existing behavior still works.

Modifying the function

We'll modify `get_full_name()` *so it handles middle names, but we'll do it in a way that breaks existing behavior.*

```
def get_full_name(first, middle, last):
    """Return a full name."""
    full_name = "{0} {1} {2}".format(first,
                                     middle, last)
    return full_name.title()
```

Using the function

```
from full_names import get_full_name

john = get_full_name('john', 'lee', 'hooker')
print(john)

david = get_full_name('david', 'lee', 'roth')
print(david)
```

A failing test (cont.)

Running the test

When you change your code, it's important to run your existing tests. This will tell you whether the changes you made affected existing behavior.

```
E
======================================================
ERROR: test_first_last (__main__.NamesTestCase)
Test names like Janis Joplin.
------------------------------------------------------
Traceback (most recent call last):
  File "test_full_names.py", line 10,
    in test_first_last
    'joplin')
TypeError: get_full_name() missing 1 required
  positional argument: 'last'

------------------------------------------------------
Ran 1 test in 0.001s

FAILED (errors=1)
```

Fixing the code

When a test fails, the code needs to be modified until the test passes again. (Don't make the mistake of rewriting your tests to fit your new code.) Here we can make the middle name optional.

```
def get_full_name(first, last, middle=''):
    """Return a full name."""
    if middle:
        full_name = "{0} {1} {2}".format(first,
                                         middle, last)
    else:
        full_name = "{0} {1}".format(first,
                                     last)
    return full_name.title()
```

Running the test

Now the test should pass again, which means our original functionality is still intact.

```
.
--------------------------------------
Ran 1 test in 0.000s

OK
```

Adding new tests

You can add as many unit tests to a test case as you need. To write a new test, add a new method to your test case class.

Testing middle names

We've shown that `get_full_name()` works for first and last names. Let's test that it works for middle names as well.

```
import unittest
from full_names import get_full_name

class NamesTestCase(unittest.TestCase):
    """Tests for names.py."""

    def test_first_last(self):
        """Test names like Janis Joplin."""
        full_name = get_full_name('janis',
                'joplin')
        self.assertEqual(full_name,
                'Janis Joplin')

    def test_middle(self):
        """Test names like David Lee Roth."""
        full_name = get_full_name('david',
                'roth', 'lee')
        self.assertEqual(full_name,
                'David Lee Roth')

unittest.main()
```

Running the tests

The two dots represent two passing tests.

```
..
--------------------------------------
Ran 2 tests in 0.000s

OK
```

A variety of assert methods

Python provides a number of assert methods you can use to test your code.

Verify that a==b, or a != b

```
assertEqual(a, b)
assertNotEqual(a, b)
```

Verify that x is True, or x is False

```
assertTrue(x)
assertFalse(x)
```

Verify an item is in a list, or not in a list

```
assertIn(item, list)
assertNotIn(item, list)
```

Testing a class

Testing a class is similar to testing a function, since you'll mostly be testing your methods.

A class to test

Save as accountant.py

```
class Accountant():
    """Manage a bank account."""

    def __init__(self, balance=0):
        self.balance = balance

    def deposit(self, amount):
        self.balance += amount

    def withdraw(self, amount):
        self.balance -= amount
```

Building a testcase

For the first test, we'll make sure we can start out with different initial balances. Save this as test_accountant.py.

```
import unittest
from accountant import Accountant

class TestAccountant(unittest.TestCase):
    """Tests for the class Accountant."""

    def test_initial_balance(self):
        # Default balance should be 0.
        acc = Accountant()
        self.assertEqual(acc.balance, 0)

        # Test non-default balance.
        acc = Accountant(100)
        self.assertEqual(acc.balance, 100)

unittest.main()
```

Running the test

```
.
--------------------------------------
Ran 1 test in 0.000s

OK
```

When is it okay to modify tests?

In general you shouldn't modify a test once it's written. When a test fails it usually means new code you've written has broken existing functionality, and you need to modify the new code until all existing tests pass.

If your original requirements have changed, it may be appropriate to modify some tests. This usually happens in the early stages of a project when desired behavior is still being sorted out.

The setUp() method

When testing a class, you usually have to make an instance of the class. The `setUp()` method is run before every test. Any instances you make in `setUp()` are available in every test you write.

Using setUp() to support multiple tests

The instance `self.acc` can be used in each new test.

```
import unittest
from accountant import Accountant

class TestAccountant(unittest.TestCase):
    """Tests for the class Accountant."""

    def setUp(self):
        self.acc = Accountant()

    def test_initial_balance(self):
        # Default balance should be 0.
        self.assertEqual(self.acc.balance, 0)

        # Test non-default balance.
        acc = Accountant(100)
        self.assertEqual(acc.balance, 100)

    def test_deposit(self):
        # Test single deposit.
        self.acc.deposit(100)
        self.assertEqual(self.acc.balance, 100)

        # Test multiple deposits.
        self.acc.deposit(100)
        self.acc.deposit(100)
        self.assertEqual(self.acc.balance, 300)

    def test_withdrawal(self):
        # Test single withdrawal.
        self.acc.deposit(1000)
        self.acc.withdraw(100)
        self.assertEqual(self.acc.balance, 900)

unittest.main()
```

Running the tests

```
...
--------------------------------------
Ran 3 tests in 0.001s

OK
```

More cheat sheets available at
ehmatthes.github.io/pcc/

Beginner's Python Cheat Sheet - Pygame

What is Pygame?

Pygame is a framework for making games using Python. Making games is fun, and it's a great way to expand your programming skills and knowledge. Pygame takes care of many of the lower-level tasks in building games, which lets you focus on the aspects of your game that make it interesting.

Installing Pygame

Pygame runs on all systems, but setup is slightly different on each OS. The instructions here assume you're using Python 3, and provide a minimal installation of Pygame. If these instructions don't work for your system, see the more detailed notes at http://ehmatthes.github.io/pcc/.

Pygame on Linux

```
$ sudo apt-get install python3-dev mercurial
    libsdl-image1.2-dev libsdl2-dev
    libsdl-ttf2.0-dev
$ pip install --user
    hg+http://bitbucket.org/pygame/pygame
```

Pygame on OS X

This assumes you've used Homebrew to install Python 3.

```
$ brew install hg sdl sdl_image sdl_ttf
$ pip install --user
    hg+http://bitbucket.org/pygame/pygame
```

Pygame on Windows

Find an installer at https://bitbucket.org/pygame/pygame/downloads/ or http://www.lfd.uci.edu/~gohlke/pythonlibs/#pygame that matches your version of Python. Run the installer file if it's a .exe or .msi file. If it's a .whl file, use pip to install Pygame:

```
> python -m pip install --user
    pygame-1.9.2a0-cp35-none-win32.whl
```

Testing your installation

To test your installation, open a terminal session and try to import Pygame. If you don't get any error messages, your installation was successful.

```
$ python
>>> import pygame
>>>
```

Starting a game

The following code sets up an empty game window, and starts an event loop and a loop that continually refreshes the screen.

An empty game window

```
import sys
import pygame as pg

def run_game():
    # Initialize and set up screen.
    pg.init()
    screen = pg.display.set_mode((1200, 800))
    pg.display.set_caption("Alien Invasion")

    # Start main loop.
    while True:
        # Start event loop.
        for event in pg.event.get():
            if event.type == pg.QUIT:
                sys.exit()

        # Refresh screen.
        pg.display.flip()

run_game()
```

Setting a custom window size

The display.set_mode() function accepts a tuple that defines the screen size.

```
screen_dim = (1200, 800)
screen = pg.display.set_mode(screen_dim)
```

Setting a custom background color

Colors are defined as a tuple of red, green, and blue values. Each value ranges from 0-255.

```
bg_color = (230, 230, 230)
screen.fill(bg_color)
```

Pygame rect objects

Many objects in a game can be treated as simple rectangles, rather than their actual shape. This simplifies code without noticeably affecting game play. Pygame has a rect object that makes it easy to work with game objects.

Getting the screen rect object

We already have a screen object; we can easily access the rect object associated with the screen.

```
screen_rect = screen.get_rect()
```

Finding the center of the screen

Rect objects have a center attribute which stores the center point.

```
screen_center = screen_rect.center
```

Pygame rect objects (cont.)

Useful rect attributes

Once you have a rect object, there are a number of attributes that are useful when positioning objects and detecting relative positions of objects. (You can find more attributes in the Pygame documentation.)

```
# Individual x and y values:
screen_rect.left, screen_rect.right
screen_rect.top, screen_rect.bottom
screen_rect.centerx, screen_rect.centery
screen_rect.width, screen_rect.height

# Tuples
screen_rect.center
screen_rect.size
```

Creating a rect object

You can create a rect object from scratch. For example a small rect object that's filled in can represent a bullet in a game. The Rect() class takes the coordinates of the upper left corner, and the width and height of the rect. The draw.rect() function takes a screen object, a color, and a rect. This function fills the given rect with the given color.

```
bullet_rect = pg.Rect(100, 100, 3, 15)
color = (100, 100, 100)
pg.draw.rect(screen, color, bullet_rect)
```

Working with images

Many objects in a game are images that are moved around the screen. It's easiest to use bitmap (.bmp) image files, but you can also configure your system to work with jpg, png, and gif files as well.

Loading an image

```
ship = pg.image.load('images/ship.bmp')
```

Getting the rect object from an image

```
ship_rect = ship.get_rect()
```

Positioning an image

With rects, it's easy to position an image wherever you want on the screen, or in relation to another object. The following code positions a ship object at the bottom center of the screen.

```
ship_rect.midbottom = screen_rect.midbottom
```

Working with images (cont.)

Drawing an image to the screen

Once an image is loaded and positioned, you can draw it to the screen with the blit() method. The blit() method acts on the screen object, and takes the image object and image rect as arguments.

```
# Draw ship to screen.
screen.blit(ship, ship_rect)
```

The blitme() method

Game objects such as ships are often written as classes. Then a blitme() method is usually defined, which draws the object to the screen.

```
def blitme(self):
    """Draw ship at current location."""
    self.screen.blit(self.image, self.rect)
```

Responding to keyboard input

Pygame watches for events such as key presses and mouse actions. You can detect any event you care about in the event loop, and respond with any action that's appropriate for your game.

Responding to key presses

Pygame's main event loop registers a KEYDOWN event any time a key is pressed. When this happens, you can check for specific keys.

```
for event in pg.event.get():
    if event.type == pg.KEYDOWN:
        if event.key == pg.K_RIGHT:
            ship_rect.x += 1
        elif event.key == pg.K_LEFT:
            ship_rect.x -= 1
        elif event.key == pg.K_SPACE:
            ship.fire_bullet()
        elif event.key == pg.K_q:
            sys.exit()
```

Responding to released keys

When the user releases a key, a KEYUP event is triggered.

```
if event.type == pg.KEYUP:
    if event.key == pg.K_RIGHT:
        ship.moving_right = False
```

Pygame documentation

The Pygame documentation is really helpful when building your own games. The home page for the Pygame project is at http://pygame.org/, and the home page for the documentation is at http://pygame.org/docs/.

The most useful part of the documentation are the pages about specific parts of Pygame, such as the Rect() class and the sprite module. You can find a list of these elements at the top of the help pages.

Responding to mouse events

Pygame's event loop registers an event any time the mouse moves, or a mouse button is pressed or released.

Responding to the mouse button

```
for event in pg.event.get():
    if event.type == pg.MOUSEBUTTONDOWN:
        ship.fire_bullet()
```

Finding the mouse position

The mouse position is returned as a tuple.

```
mouse_pos = pg.mouse.get_pos()
```

Clicking a button

You might want to know if the cursor is over an object such as a button. The rect.collidepoint() method returns true when a point is inside a rect object.

```
if button_rect.collidepoint(mouse_pos):
    start_game()
```

Hiding the mouse

```
pg.mouse.set_visible(False)
```

Pygame groups

Pygame has a Group class which makes working with a group of similar objects easier. A group is like a list, with some extra functionality that's helpful when building games.

Making and filling a group

An object that will be placed in a group must inherit from Sprite.

```
from pygame.sprite import Sprite, Group

def Bullet(Sprite):
    ...
    def draw_bullet(self):
        ...
    def update(self):
        ...

bullets = Group()

new_bullet = Bullet()
bullets.add(new_bullet)
```

Looping through the items in a group

The sprites() method returns all the members of a group.

```
for bullet in bullets.sprites():
    bullet.draw_bullet()
```

Calling update() on a group

Calling update() on a group automatically calls update() on each member of the group.

```
bullets.update()
```

Pygame groups (cont.)

Removing an item from a group

It's important to delete elements that will never appear again in the game, so you don't waste memory and resources.

```
bullets.remove(bullet)
```

Detecting collisions

You can detect when a single object collides with any member of a group. You can also detect when any member of one group collides with a member of another group.

Collisions between a single object and a group

The spritecollideany() function takes an object and a group, and returns True if the object overlaps with any member of the group.

```
if pg.sprite.spritecollideany(ship, aliens):
    ships_left -= 1
```

Collisions between two groups

The sprite.groupcollide() function takes two groups, and two booleans. The function returns a dictionary containing information about the members that have collided. The booleans tell Pygame whether to delete the members of either group that have collided.

```
collisions = pg.sprite.groupcollide(
        bullets, aliens, True, True)

score += len(collisions) * alien_point_value
```

Rendering text

You can use text for a variety of purposes in a game. For example you can share information with players, and you can display a score.

Displaying a message

The following code defines a message, then a color for the text and the background color for the message. A font is defined using the default system font, with a font size of 48. The font.render() function is used to create an image of the message, and we get the rect object associated with the image. We then center the image on the screen and display it.

```
msg = "Play again?"
msg_color = (100, 100, 100)
bg_color = (230, 230, 230)

f = pg.font.SysFont(None, 48)
msg_image = f.render(msg, True, msg_color,
        bg_color)
msg_image_rect = msg_image.get_rect()
msg_image_rect.center = screen_rect.center
screen.blit(msg_image, msg_image_rect)
```

More cheat sheets available at
ehmatthes.github.io/pcc/

Beginner's Python Cheat Sheet — matplotlib

What is matplotlib?

Data visualization involves exploring data through visual representations. The matplotlib package helps you make visually appealing representations of the data you're working with. matplotlib is extremely flexible; these examples will help you get started with a few simple visualizations.

Installing matplotlib

matplotlib runs on all systems, but setup is slightly different depending on your OS. If the minimal instructions here don't work for you, see the more detailed instructions at http://ehmatthes.github.io/pcc/. You should also consider installing the Anaconda distrubution of Python from https://continuum.io/downloads/, which includes matplotlib.

matplotlib on Linux

```
$ sudo apt-get install python3-matplotlib
```

matplotlib on OS X

Start a terminal session and enter `import matplotlib` *to see if it's already installed on your system. If not, try this command:*

```
$ pip install --user matplotlib
```

matplotlib on Windows

You first need to install Visual Studio, which you can do from https://dev.windows.com/. The Community edition is free. Then go to https://pypi.python.org/pypi/matplotlib/ or http://www.lfd.uic.edu/~gohlke/pythonlibs/#matplotlib and download an appropriate installer file.

Line graphs and scatter plots

Making a line graph

```
import matplotlib.pyplot as plt

x_values = [0, 1, 2, 3, 4, 5]
squares = [0, 1, 4, 9, 16, 25]
plt.plot(x_values, squares)
plt.show()
```

Line graphs and scatter plots (cont.)

Making a scatter plot

The scatter() function takes a list of x values and a list of y values, and a variety of optional arguments. The `s=10` *argument controls the size of each point.*

```
import matplotlib.pyplot as plt

x_values = list(range(1000))
squares = [x**2 for x in x_values]

plt.scatter(x_values, squares, s=10)
plt.show()
```

Customizing plots

Plots can be customized in a wide variety of ways. Just about any element of a plot can be customized.

Adding titles and labels, and scaling axes

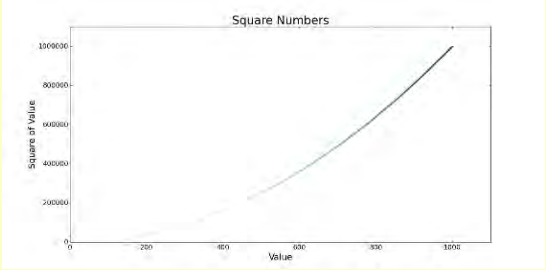
```
import matplotlib.pyplot as plt

x_values = list(range(1000))
squares = [x**2 for x in x_values]
plt.scatter(x_values, squares, s=10)

plt.title("Square Numbers", fontsize=24)
plt.xlabel("Value", fontsize=18)
plt.ylabel("Square of Value", fontsize=18)
plt.tick_params(axis='both', which='major',
                labelsize=14)
plt.axis([0, 1100, 0, 1100000])

plt.show()
```

Using a colormap

A colormap varies the point colors from one shade to another, based on a certain value for each point. The value used to determine the color of each point is passed to the `c` *argument, and the* `cmap` *argument specifies which colormap to use.*

The `edgecolor='none'` *argument removes the black outline from each point.*

```
plt.scatter(x_values, squares, c=squares,
        cmap=plt.cm.Blues, edgecolor='none',
        s=10)
```



Customizing plots (cont.)

Emphasizing points

You can plot as much data as you want on one plot. Here we re-plot the first and last points larger to emphasize them.

```
import matplotlib.pyplot as plt

x_values = list(range(1000))
squares = [x**2 for x in x_values]
plt.scatter(x_values, squares, c=squares,
        cmap=plt.cm.Blues, edgecolor='none',
        s=10)

plt.scatter(x_values[0], squares[0], c='green',
        edgecolor='none', s=100)
plt.scatter(x_values[-1], squares[-1], c='red',
        edgecolor='none', s=100)

plt.title("Square Numbers", fontsize=24)
--snip--
```

Removing axes

You can customize or remove axes entirely. Here's how to access each axis, and hide it.

```
plt.axes().get_xaxis().set_visible(False)
plt.axes().get_yaxis().set_visible(False)
```

Setting a custom figure size

You can make your plot as big or small as you want. Before plotting your data, add the following code. The `dpi` *argument is optional; if you don't know your system's resolution you can omit the argument and adjust the* `figsize` *argument accordingly.*

```
plt.figure(dpi=128, figsize=(10, 6))
```

Saving a plot

The matplotlib viewer has an interactive save button, but you can also save your visualizations programmatically. To do so, replace `plt.show()` *with* `plt.savefig()`*. The* `bbox_inches='tight'` *argument trims extra whitespace from the plot.*

```
plt.savefig('squares.png', bbox_inches='tight')
```

Online resources

The matplotlib gallery and documentation are at http://matplotlib.org/. Be sure to visit the examples, gallery, and pyplot links.

Multiple plots

You can make as many plots as you want on one figure. When you make multiple plots, you can emphasize relationships in the data. For example you can fill the space between two sets of data.

Plotting two sets of data

Here we use `plt.scatter()` *twice to plot square numbers and cubes on the same figure.*

```
import matplotlib.pyplot as plt

x_values = list(range(11))
squares = [x**2 for x in x_values]
cubes = [x**3 for x in x_values]

plt.scatter(x_values, squares, c='blue',
        edgecolor='none', s=20)
plt.scatter(x_values, cubes, c='red',
        edgecolor='none', s=20)

plt.axis([0, 11, 0, 1100])
plt.show()
```

Filling the space between data sets

The `fill_between()` *method fills the space between two data sets. It takes a series of x-values and two series of y-values. It also takes a* `facecolor` *to use for the fill, and an optional* `alpha` *argument that controls the color's transparency.*

```
plt.fill_between(x_values, cubes, squares,
        facecolor='blue', alpha=0.25)
```

Working with dates and times

Many interesting data sets have a date or time as the x-value. Python's datetime module helps you work with this kind of data.

Generating the current date

The `datetime.now()` *function returns a datetime object representing the current date and time.*

```
from datetime import datetime as dt

today = dt.now()
date_string = dt.strftime(today, '%m/%d/%Y')
print(date_string)
```

Generating a specific date

You can also generate a datetime object for any date and time you want. The positional order of arguments is year, month, and day. The hour, minute, second, and microsecond arguments are optional.

```
from datetime import datetime as dt

new_years = dt(2017, 1, 1)
fall_equinox = dt(year=2016, month=9, day=22)
```

Working with dates and times (cont.)

Datetime formatting arguments

The `strftime()` *function generates a formatted string from a datetime object, and the* `strptime()` *function genereates a datetime object from a string. The following codes let you work with dates exactly as you need to.*

```
%A    Weekday name, such as Monday
%B    Month name, such as January
%m    Month, as a number (01 to 12)
%d    Day of the month, as a number (01 to 31)
%Y    Four-digit year, such as 2016
%y    Two-digit year, such as 16
%H    Hour, in 24-hour format (00 to 23)
%I    Hour, in 12-hour format (01 to 12)
%p    AM or PM
%M    Minutes (00 to 59)
%S    Seconds (00 to 61)
```

Converting a string to a datetime object

```
new_years = dt.strptime('1/1/2017', '%m/%d/%Y')
```

Converting a datetime object to a string

```
ny_string = dt.strftime(new_years, '%B %d, %Y')
print(ny_string)
```

Plotting high temperatures

The following code creates a list of dates and a corresponding list of high temperatures. It then plots the high temperatures, with the date labels displayed in a specific format.

```
from datetime import datetime as dt

import matplotlib.pyplot as plt
from matplotlib import dates as mdates

dates = [
    dt(2016, 6, 21), dt(2016, 6, 22),
    dt(2016, 6, 23), dt(2016, 6, 24),
    ]

highs = [57, 68, 64, 59]

fig = plt.figure(dpi=128, figsize=(10,6))
plt.plot(dates, highs, c='red')
plt.title("Daily High Temps", fontsize=24)
plt.ylabel("Temp (F)", fontsize=16)

x_axis = plt.axes().get_xaxis()
x_axis.set_major_formatter(
        mdates.DateFormatter('%B %d %Y')
        )
fig.autofmt_xdate()

plt.show()
```

Multiple plots in one figure

You can include as many individual graphs in one figure as you want. This is useful, for example, when comparing related datasets.

Sharing an x-axis

The following code plots a set of squares and a set of cubes on two separate graphs that share a common x-axis.

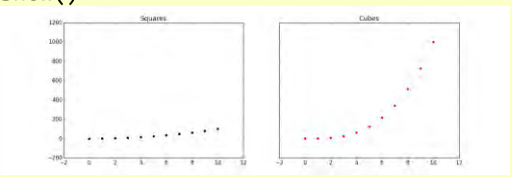
The `plt.subplots()` *function returns a figure object and a tuple of axes. Each set of axes corresponds to a separate plot in the figure. The first two arguments control the number of rows and columns generated in the figure.*

```
import matplotlib.pyplot as plt

x_vals = list(range(11))
squares = [x**2 for x in x_vals]
cubes = [x**3 for x in x_vals]

fig, axarr = plt.subplots(2, 1, sharex=True)

axarr[0].scatter(x_vals, squares)
axarr[0].set_title('Squares')

axarr[1].scatter(x_vals, cubes, c='red')
axarr[1].set_title('Cubes')

plt.show()
```

Sharing a y-axis

To share a y-axis, we use the `sharey=True` *argument.*

```
import matplotlib.pyplot as plt

x_vals = list(range(11))
squares = [x**2 for x in x_vals]
cubes = [x**3 for x in x_vals]

fig, axarr = plt.subplots(1, 2, sharey=True)

axarr[0].scatter(x_vals, squares)
axarr[0].set_title('Squares')

axarr[1].scatter(x_vals, cubes, c='red')
axarr[1].set_title('Cubes')

plt.show()
```



More cheat sheets available at
ehmatthes.github.io/pcc/

Beginner's Python Cheat Sheet — Pygal

What is Pygal?

Data visualization involves exploring data through visual representations. Pygal helps you make visually appealing representations of the data you're working with. Pygal is particularly well suited for visualizations that will be presented online, because it supports interactive elements.

Installing Pygal

Pygal can be installed using pip.

Pygal on Linux and OS X

```
$ pip install --user pygal
```

Pygal on Windows

```
> python -m pip install --user pygal
```

Line graphs, scatter plots, and bar graphs

To make a plot with Pygal, you specify the kind of plot and then add the data.

Making a line graph

To view the output, open the file squares.svg in a browser.

```
import pygal

x_values = [0, 1, 2, 3, 4, 5]
squares = [0, 1, 4, 9, 16, 25]

chart = pygal.Line()
chart.force_uri_protocol = 'http'
chart.add('x^2', squares)
chart.render_to_file('squares.svg')
```

Adding labels and a title

```
--snip--
chart = pygal.Line()
chart.force_uri_protocol = 'http'
chart.title = "Squares"
chart.x_labels = x_values
chart.x_title = "Value"
chart.y_title = "Square of Value"
chart.add('x^2', squares)
chart.render_to_file('squares.svg')
```

Line graphs, scatter plots, and bar graphs (cont.)

Making a scatter plot

The data for a scatter plot needs to be a list containing tuples of the form (x, y). The `stroke=False` argument tells Pygal to make an XY chart with no line connecting the points.

```
import pygal

squares = [
    (0, 0), (1, 1), (2, 4), (3, 9),
    (4, 16), (5, 25),
    ]

chart = pygal.XY(stroke=False)
chart.force_uri_protocol = 'http'
chart.add('x^2', squares)
chart.render_to_file('squares.svg')
```

Using a list comprehension for a scatter plot

A list comprehension can be used to effficiently make a dataset for a scatter plot.

```
squares = [(x, x**2) for x in range(1000)]
```

Making a bar graph

A bar graph requires a list of values for the bar sizes. To label the bars, pass a list of the same length to `x_labels`.

```
import pygal

outcomes = [1, 2, 3, 4, 5, 6]
frequencies = [18, 16, 18, 17, 18, 13]

chart = pygal.Bar()
chart.force_uri_protocol = 'http'
chart.x_labels = outcomes
chart.add('D6', frequencies)
chart.render_to_file('rolling_dice.svg')
```

Making a bar graph from a dictionary

Since each bar needs a label and a value, a dictionary is a great way to store the data for a bar graph. The keys are used as the labels along the x-axis, and the values are used to determine the height of each bar.

```
import pygal

results = {
    1:18, 2:16, 3:18,
    4:17, 5:18, 6:13,
    }

chart = pygal.Bar()
chart.force_uri_protocol = 'http'
chart.x_labels = results.keys()
chart.add('D6', results.values())
chart.render_to_file('rolling_dice.svg')
```

Multiple plots

You can add as much data as you want when making a visualization.

Plotting squares and cubes

```
import pygal

x_values = list(range(11))
squares = [x**2 for x in x_values]
cubes = [x**3 for x in x_values]

chart = pygal.Line()
chart.force_uri_protocol = 'http'
chart.title = "Squares and Cubes"
chart.x_labels = x_values

chart.add('Squares', squares)
chart.add('Cubes', cubes)
chart.render_to_file('squares_cubes.svg')
```

Filling the area under a data series

Pygal allows you to fill the area under or over each series of data. The default is to fill from the x-axis up, but you can fill from any horizontal line using the `zero` argument.

```
chart = pygal.Line(fill=True, zero=0)
```

Online resources

The documentation for Pygal is available at http://www.pygal.org/.

Enabling interactive features

If you're viewing svg output in a browser, Pygal needs to render the output file in a specific way. The `force_uri_protocol` attribute for chart objects needs to be set to `'http'`.

Styling plots

Pygal lets you customize many elements of a plot. There are some excellent default themes, and many options for styling individual plot elements.

Using built-in styles

To use built-in styles, import the style and make an instance of the style class. Then pass the style object with the `style` argument when you make the chart object.

```
import pygal
from pygal.style import LightGreenStyle

x_values = list(range(11))
squares = [x**2 for x in x_values]
cubes = [x**3 for x in x_values]

chart_style = LightGreenStyle()
chart = pygal.Line(style=chart_style)
chart.force_uri_protocol = 'http'
chart.title = "Squares and Cubes"
chart.x_labels = x_values

chart.add('Squares', squares)
chart.add('Cubes', cubes)
chart.render_to_file('squares_cubes.svg')
```

Parametric built-in styles

Some built-in styles accept a custom color, then generate a theme based on that color.

```
from pygal.style import LightenStyle

--snip--
chart_style = LightenStyle('#336688')
chart = pygal.Line(style=chart_style)
--snip--
```

Customizing individual style properties

Style objects have a number of properties you can set individually.

```
chart_style = LightenStyle('#336688')
chart_style.plot_background = '#CCCCCC'
chart_style.major_label_font_size = 20
chart_style.label_font_size = 16
--snip--
```

Custom style class

You can start with a bare style class, and then set only the properties you care about.

```
chart_style = Style()
chart_style.colors = [
    '#CCCCCC', '#AAAAAA', '#888888']
chart_style.plot_background = '#EEEEEE'

chart = pygal.Line(style=chart_style)
--snip--
```

Styling plots (cont.)

Configuration settings

Some settings are controlled by a `Config` object.

```
my_config = pygal.Config()
my_config.show_y_guides = False
my_config.width = 1000
my_config.dots_size = 5

chart = pygal.Line(config=my_config)
--snip--
```

Styling series

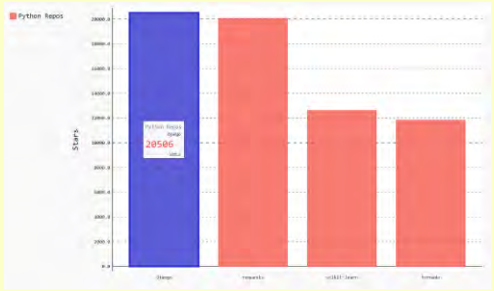
You can give each series on a chart different style settings.

```
chart.add('Squares', squares, dots_size=2)
chart.add('Cubes', cubes, dots_size=3)
```

Styling individual data points

You can style individual data points as well. To do so, write a dictionary for each data point you want to customize. A `'value'` key is required, and other properies are optional.

```
import pygal

repos = [
    {
        'value': 20506,
        'color': '#3333CC',
        'xlink': 'http://djangoproject.com/',
    },
    20054,
    12607,
    11827,
    ]

chart = pygal.Bar()
chart.force_uri_protocol = 'http'
chart.x_labels = [
    'django', 'requests', 'scikit-learn',
    'tornado',
    ]
chart.y_title = 'Stars'
chart.add('Python Repos', repos)
chart.render_to_file('python_repos.svg')
```



Plotting global datasets

Pygal can generate world maps, and you can add any data you want to these maps. Data is indicated by coloring, by labels, and by tooltips that show data when users hover over each country on the map.

Installing the world map module

The world map module is not included by default in Pygal 2.0. It can be installed with pip:

```
$ pip install --user pygal_maps_world
```

Making a world map

The following code makes a simple world map showing the countries of North America.

```
from pygal.maps.world import World

wm = World()
wm.force_uri_protocol = 'http'
wm.title = 'North America'
wm.add('North America', ['ca', 'mx', 'us'])

wm.render_to_file('north_america.svg')
```

Showing all the country codes

In order to make maps, you need to know Pygal's country codes. The following example will print an alphabetical list of each country and its code.

```
from pygal.maps.world import COUNTRIES

for code in sorted(COUNTRIES.keys()):
    print(code, COUNTRIES[code])
```

Plotting numerical data on a world map

To plot numerical data on a map, pass a dictionary to `add()` instead of a list.

```
from pygal.maps.world import World

populations = {
    'ca': 34126000,
    'us': 309349000,
    'mx': 113423000,
    }

wm = World()
wm.force_uri_protocol = 'http'
wm.title = 'Population of North America'
wm.add('North America', populations)

wm.render_to_file('na_populations.svg')
```

More cheat sheets available at
ehmatthes.github.io/pcc/

Beginner's Python Cheat Sheet — Django

What is Django?

Django is a web framework which helps you build interactive websites using Python. With Django you define the kind of data your site needs to work with, and you define the ways your users can work with that data.

Installing Django

It's usualy best to install Django to a virtual environment, where your project can be isolated from your other Python projects. Most commands assume you're working in an active virtual environment.

Create a virtual environment

```
$ python -m venv ll_env
```

Activate the environment (Linux and OS X)

```
$ source ll_env/bin/activate
```

Activate the environment (Windows)

```
> ll_env\Scripts\activate
```

Install Django to the active environment

```
(ll_env)$ pip install Django
```

Creating a project

To start a project we'll create a new project, create a database, and start a development server.

Create a new project

```
$ django-admin.py startproject learning_log .
```

Create a database

```
$ python manage.py migrate
```

View the project

After issuing this command, you can view the project at http://localhost:8000/.

```
$ python manage.py runserver
```

Create a new app

A Django project is made up of one or more apps.

```
$ python manage.py startapp learning_logs
```

Working with models

The data in a Django project is structured as a set of models.

Defining a model

To define the models for your app, modify the file models.py that was created in your app's folder. The __str__() method tells Django how to represent data objects based on this model.

```
from django.db import models

class Topic(models.Model):
    """A topic the user is learning about."""
    text = models.CharField(max_length=200)
    date_added = models.DateTimeField(
            auto_now_add=True)

    def __str__(self):
        return self.text
```

Activating a model

To use a model the app must be added to the tuple `INSTALLED_APPS`*, which is stored in the project's settings.py file.*

```
INSTALLED_APPS = (
    --snip--
    'django.contrib.staticfiles',

    # My apps
    'learning_logs',
    )
```

Migrating the database

The database needs to be modified to store the kind of data that the model represents.

```
$ python manage.py makemigrations learning_logs
$ python manage.py migrate
```

Creating a superuser

A superuser is a user account that has access to all aspects of the project.

```
$ python manage.py createsuperuser
```

Registering a model

You can register your models with Django's admin site, which makes it easier to work with the data in your project. To do this, modify the app's admin.py file. View the admin site at http://localhost:8000/admin/.

```
from django.contrib import admin

from learning_logs.models import Topic

admin.site.register(Topic)
```

Building a simple home page

Users interact with a project through web pages, and a project's home page can start out as a simple page with no data. A page usually needs a URL, a view, and a template.

Mapping a project's URLs

The project's main urls.py file tells Django where to find the urls.py files associated with each app in the project.

```
from django.conf.urls import include, url
from django.contrib import admin

urlpatterns = [
    url(r'^admin/', include(admin.site.urls)),
    url(r'', include('learning_logs.urls',
            namespace='learning_logs')),
    ]
```

Mapping an app's URLs

An app's urls.py file tells Django which view to use for each URL in the app. You'll need to make this file yourself, and save it in the app's folder.

```
from django.conf.urls import url

from . import views

urlpatterns = [
    url(r'^$', views.index, name='index'),
    ]
```

Writing a simple view

A view takes information from a request and sends data to the browser, often through a template. View functions are stored in an app's views.py file. This simple view function doesn't pull in any data, but it uses the template index.html to render the home page.

```
from django.shortcuts import render

def index(request):
    """The home page for Learning Log."""
    return render(request,
            'learning_logs/index.html')
```

Online resources

The documentation for Django is available at http://docs.djangoproject.com/. The Django documentation is thorough and user-friendly, so check it out!

Building a simple home page (cont.)

Writing a simple template

A template sets up the structure for a page. It's a mix of html and template code, which is like Python but not as powerful. Make a folder called templates inside the project folder. Inside the templates folder make another folder with the same name as the app. This is where the template files should be saved.

```
<p>Learning Log</p>

<p>Learning Log helps you keep track of your
learning, for any topic you're learning
about.</p>
```

Template inheritance

Many elements of a web page are repeated on every page in the site, or every page in a section of the site. By writing one parent template for the site, and one for each section, you can easily modify the look and feel of your entire site.

The parent template

The parent template defines the elements common to a set of pages, and defines blocks that will be filled by individual pages.

```
<p>
  <a href="{% url 'learning_logs:index' %}">
    Learning Log
  </a>
</p>

{% block content %}{% endblock content %}
```

The child template

The child template uses the `{% extends %}` *template tag to pull in the structure of the parent template. It then defines the content for any blocks defined in the parent template.*

```
{% extends 'learning_logs/base.html' %}

{% block content %}
  <p>
    Learning Log helps you keep track
    of your learning, for any topic you're
    learning about.
  </p>
{% endblock content %}
```

Template indentation

Python code is usually indented by four spaces. In templates you'll often see two spaces used for indentation, because elements tend to be nested more deeply in templates.

Another model

A new model can use an existing model. The ForeignKey attribute establishes a connection between instances of the two related models. Make sure to migrate the database after adding a new model to your app.

Defining a model with a foreign key

```
class Entry(models.Model):
    """Learning log entries for a topic."""
    topic = models.ForeignKey(Topic)
    text = models.TextField()
    date_added = models.DateTimeField(
            auto_now_add=True)

    def __str__(self):
        return self.text[:50] + "..."
```

Building a page with data

Most pages in a project need to present data that's specific to the current user.

URL parameters

A URL often needs to accept a parameter telling it which data to access from the database. The second URL pattern shown here looks for the ID of a specific topic and stores it in the parameter `topic_id`.

```
urlpatterns = [
    url(r'^$', views.index, name='index'),
    url(r'^topics/(?P<topic_id>\d+)/$',
            views.topic, name='topic'),
    ]
```

Using data in a view

The view uses a parameter from the URL to pull the correct data from the database. In this example the view is sending a `context` *dictionary to the template, containing data that should be displayed on the page.*

```
def topic(request, topic_id):
    """Show a topic and all its entries."""
    topic = Topics.objects.get(id=topic_id)
    entries = topic.entry_set.order_by(
            '-date_added')
    context = {
        'topic': topic,
        'entries': entries,
        }
    return render(request,
        'learning_logs/topic.html', context)
```

Restarting the development server

If you make a change to your project and the change doesn't seem to have any effect, try restarting the server:

```
$ python manage.py runserver
```

Building a page with data (cont.)

Using data in a template

The data in the view function's `context` *dictionary is available within the template. This data is accessed using template variables, which are indicated by doubled curly braces.*

The vertical line after a template variable indicates a filter. In this case a filter called `date` *formats date objects, and the filter* `linebreaks` *renders paragraphs properly on a web page.*

```
{% extends 'learning_logs/base.html' %}

{% block content %}

  <p>Topic: {{ topic }}</p>

  <p>Entries:</p>
  <ul>
  {% for entry in entries %}
    <li>
    <p>
      {{ entry.date_added|date:'M d, Y H:i' }}
    </p>
    <p>
      {{ entry.text|linebreaks }}
    </p>
    </li>
  {% empty %}
    <li>There are no entries yet.</li>
  {% endfor %}
  </ul>

{% endblock content %}
```

The Django shell

You can explore the data in your project from the command line. This is helpful for developing queries and testing code snippets.

Start a shell session

```
$ python manage.py shell
```

Access data from the project

```
>>> from learning_logs.models import Topic
>>> Topic.objects.all()
[<Topic: Chess>, <Topic: Rock Climbing>]
>>> topic = Topic.objects.get(id=1)
>>> topic.text
'Chess'
```

More cheat sheets available at
ehmatthes.github.io/pcc/

Beginner's Python Cheat Sheet — Django, Part 2

Users and forms

Most web applications need to let users create accounts. This lets users create and work with their own data. Some of this data may be private, and some may be public. Django's forms allow users to enter and modify their data.

User accounts

User accounts are handled by a dedicated app called `users`. Users need to be able to register, log in, and log out. Django automates much of this work for you.

Making a users app

After making the app, be sure to add `'users'` to `INSTALLED_APPS` in the project's settings.py file.

```
$ python manage.py startapp users
```

Including URLS for the users app

Add a line to the project's urls.py file so the `users` app's URLs are included in the project.

```
urlpatterns = [
    url(r'^admin/', include(admin.site.urls)),
    url(r'^users/', include('users.urls',
            namespace='users')),
    url(r'', include('learning_logs.urls',
            namespace='learning_logs')),
    ]
```

Using forms in Django

There are a number of ways to create forms and work with them. You can use Django's defaults, or completely customize your forms. For a simple way to let users enter data based on your models, use a `ModelForm`. This creates a form that allows users to enter data that will populate the fields on a model.

The register view on the back of this sheet shows a simple approach to form processing. If the view doesn't receive data from a form, it responds with a blank form. If it receives `POST` data from a form, it validates the data and then saves it to the database.

User accounts (cont.)

Defining the URLs

Users will need to be able to log in, log out, and register. Make a new urls.py file in the `users` app folder. The login view is a default view provided by Django.

```
from django.conf.urls import url
from django.contrib.auth.views import login

from . import views

urlpatterns = [
    url(r'^login/$', login,
        {'template_name': 'users/login.html'},
        name='login'),
    url(r'^logout/$', views.logout_view,
        name='logout'),
    url(r'^register/$', views.register,
        name='register'),
    ]
```

The login template

The login view is provided by default, but you need to provide your own login template. The template shown here displays a simple login form, and provides basic error messages. Make a templates folder in the users folder, and then make a users folder in the templates folder. Save this file as login.html.

The tag `{% csrf_token %}` helps prevent a common type of attack with forms. The `{{ form.as_p }}` element displays the default login form in paragraph format. The `<input>` element named `next` redirects the user to the home page after a successful login.

```
{% extends "learning_logs/base.html" %}

{% block content %}
  {% if form.errors %}
  <p>
    Your username and password didn't match.
    Please try again.
  </p>
  {% endif %}

  <form method="post"
      action="{% url 'users:login' %}">
    {% csrf token %}
    {{ form.as_p }}
    <button name="submit">log in</button>

    <input type="hidden" name="next"
      value="{% url 'learning_logs:index' %}"/>
  </form>

{% endblock content %}
```

User accounts (cont.)

Showing the current login status

You can modify the base.html template to show whether the user is currently logged in, and to provide a link to the login and logout pages. Django makes a `user` object available to every template, and this template takes advantage of this object.

The `user.is_authenticated` tag allows you to serve specific content to users depending on whether they have logged in or not. The `{{ user.username }}` property allows you to greet users who have logged in. Users who haven't logged in see links to register or log in.

```
<p>
  <a href="{% url 'learning_logs:index' %}">
    Learning Log
  </a>
  {% if user.is_authenticated %}
    Hello, {{ user.username }}.
    <a href="{% url 'users:logout' %}">
      log out
    </a>
  {% else %}
    <a href="{% url 'users:register' %}">
      register
    </a> -
    <a href="{% url 'users:login' %}">
      log in
    </a>
  {% endif %}
</p>

{% block content %}{% endblock content %}
```

The logout view

The `logout_view()` function uses Django's `logout()` function and then redirects the user back to the home page. Since there is no logout page, there is no logout template. Make sure to write this code in the views.py file that's stored in the users app folder.

```
from django.http import HttpResponseRedirect
from django.core.urlresolvers import reverse
from django.contrib.auth import logout

def logout_view(request):
    """Log the user out."""
    logout(request)
    return HttpResponseRedirect(
        reverse('learning_logs:index'))
```

User accounts (cont.)

The register view

The register view needs to display a blank registration form when the page is first requested, and then process completed registration forms. A successful registration logs the user in and redirects to the home page.

```
from django.contrib.auth import login
from django.contrib.auth import authenticate
from django.contrib.auth.forms import \
        UserCreationForm

def register(request):
    """Register a new user."""
    if request.method != 'POST':
        # Show blank registration form.
        form = UserCreationForm()
    else:
        # Process completed form.
        form = UserCreationForm(
                data=request.POST)

        if form.is_valid():
            new_user = form.save()
            # Log in, redirect to home page.
            pw = request.POST['password1']
            authenticated_user = authenticate(
                username=new_user.username,
                password=pw
            )
            login(request, authenticated_user)
            return HttpResponseRedirect(
                reverse('learning_logs:index'))

    context = {'form': form}
    return render(request,
        'users/register.html', context)
```

Styling your project

The django-bootstrap3 app allows you to use the Bootstrap library to make your project look visually appealing. The app provides tags that you can use in your templates to style individual elements on a page. Learn more at http://django-bootstrap3.readthedocs.io/.

Deploying your project

Heroku lets you push your project to a live server, making it available to anyone with an internet connection. Heroku offers a free service level, which lets you learn the deployment process without any commitment. You'll need to install a set of heroku tools, and use git to track the state of your project. See http://devcenter.heroku.com/, and click on the Python link.

User accounts (cont.)

The register template

The register template displays the registration form in paragraph formats.

```
{% extends 'learning_logs/base.html' %}

{% block content %}

  <form method='post'
      action="{% url 'users:register' %}">

    {% csrf_token %}
    {{ form.as_p }}

    <button name='submit'>register</button>
    <input type='hidden' name='next'
      value="{% url 'learning_logs:index' %}"/>

  </form>

{% endblock content %}
```

Connecting data to users

Users will have data that belongs to them. Any model that should be connected directly to a user needs a field connecting instances of the model to a specific user.

Making a topic belong to a user

Only the highest-level data in a hierarchy needs to be directly connected to a user. To do this import the `User` model, and add it as a foreign key on the data model.

After modifying the model you'll need to migrate the database. You'll need to choose a user ID to connect each existing instance to.

```
from django.db import models
from django.contrib.auth.models import User

class Topic(models.Model):
    """A topic the user is learning about."""
    text = models.CharField(max_length=200)
    date_added = models.DateTimeField(
            auto_now_add=True)
    owner = models.ForeignKey(User)

    def __str__(self):
        return self.text
```

Querying data for the current user

In a view, the request object has a `user` attribute. You can use this attribute to query for the user's data. The `filter()` function then pulls the data that belongs to the current user.

```
topics = Topic.objects.filter(
                owner=request.user)
```

Connecting data to users (cont.)

Restricting access to logged-in users

Some pages are only relevant to registered users. The views for these pages can be protected by the `@login_required` decorator. Any view with this decorator will automatically redirect non-logged in users to an appropriate page. Here's an example views.py file.

```
from django.contrib.auth.decorators import /
        login_required
--snip--

@login_required
def topic(request, topic_id):
    """Show a topic and all its entries."""
```

Setting the redirect URL

The `@login_required` decorator sends unauthorized users to the login page. Add the following line to your project's settings.py file so Django will know how to find your login page.

```
LOGIN_URL = '/users/login/'
```

Preventing inadvertent access

Some pages serve data based on a parameter in the URL. You can check that the current user owns the requested data, and return a 404 error if they don't. Here's an example view.

```
from django.http import Http404

--snip--
def topic(request, topic_id):
    """Show a topic and all its entries."""
    topic = Topics.objects.get(id=topic_id)
    if topic.owner != request.user:
        raise Http404
    --snip--
```

Using a form to edit data

If you provide some initial data, Django generates a form with the user's existing data. Users can then modify and save their data.

Creating a form with initial data

The instance parameter allows you to specify initial data for a form.

```
form = EntryForm(instance=entry)
```

Modifying data before saving

The argument `commit=False` allows you to make changes before writing data to the database.

```
new_topic = form.save(commit=False)
new_topic.owner = request.user
new_topic.save()
```

More cheat sheets available at
ehmatthes.github.io/pcc/